# Operating System Concepts

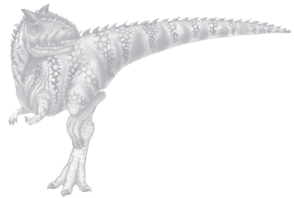
TENTH EDITION

ABRAHAM SILBERSCHATZ • PETER BAER GALVIN • GREG GAGNE

WILEY

# OPERATING SYSTEM CONCEPTS

TENTH EDITION



# OPERATING SYSTEM CONCEPTS

ABRAHAM SILBERSCHATZ
Yale University

PETER BAER GALVIN
Cambridge Computer and Starfish Storage

GREG GAGNE
Westminster College

TENTH EDITION

WILEY

| | |
|---|---|
| Publisher | Laurie Rosatone |
| Editorial Director | Don Fowley |
| Development Editor | Ryann Dannelly |
| Freelance Developmental Editor | Chris Nelson/Factotum |
| Executive Marketing Manager | Glenn Wilson |
| Senior Content Manage | Valerie Zaborski |
| Senior Production Editor | Ken Santor |
| Media Specialist | Ashley Patterson |
| Editorial Assistant | Anna Pham |
| Cover Designer | Tom Nery |
| Cover art | © metha189/Shutterstock |

This book was set in Palatino by the author using LaTeX and printed and bound by LSC Kendallville. The cover was printed by LSC Kendallville.

Copyright © 2018, 2013, 2012, 2008 John Wiley & Sons, Inc. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc. 222 Rosewood Drive, Danvers, MA 01923, (978)750-8400, fax (978)750-4470. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030 (201)748-6011, fax (201)748-6008, E-Mail: PERMREQ@WILEY.COM.

Evaluation copies are provided to qualified academics and professionals for review purposes only, for use in their courses during the next academic year. These copies are licensed and may not be sold or transferred to a third party. Upon completion of the review period, please return the evaluation copy to Wiley. Return instructions and a free-of-charge return shipping label are available at www.wiley.com/go/evalreturn. Outside of the United States, please contact your local representative.

Library of Congress Cataloging-in-Publication Data

Names: Silberschatz, Abraham, author. | Galvin, Peter B., author. | Gagne,
Greg, author.
Title: Operating system concepts / Abraham Silberschatz, Yale University,
Peter Baer Galvin, Pluribus Networks, Greg Gagne, Westminster College.
Description: 10th edition. | Hoboken, NJ : Wiley, [2018] | Includes
bibliographical references and index. |
Identifiers: LCCN 2017043464 (print) | LCCN 2017045986 (ebook) | ISBN
9781119320913 (enhanced ePub)
Subjects: LCSH: Operating systems (Computers)
Classification: LCC QA76.76.O63 (ebook) | LCC QA76.76.O63 S55825 2018 (print)
| DDC 005.4/3--dc23
LC record available at https://lccn.loc.gov/2017043464

The inside back cover will contain printing identification and country of origin if omitted from this page. In addition, if the ISBN on the back cover differs from the ISBN on this page, the one on the back cover is correct.

Enhanced ePub ISBN 978-1-119-32091-3

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

*To my children, Lemor, Sivan, and Aaron*
*and my Nicolette*

*Avi Silberschatz*

*To my wife, Carla,*
*and my children, Gwen, Owen, and Maddie*

*Peter Baer Galvin*

*To my wife, Pat,*
*and our sons, Tom and Jay*

*Greg Gagne*

# Preface

Operating systems are an essential part of any computer system. Similarly, a course on operating systems is an essential part of any computer science education. This field is undergoing rapid change, as computers are now prevalent in virtually every arena of day-to-day life—from embedded devices in automobiles through the most sophisticated planning tools for governments and multinational firms. Yet the fundamental concepts remain fairly clear, and it is on these that we base this book.

We wrote this book as a text for an introductory course in operating systems at the junior or senior undergraduate level or at the first-year graduate level. We hope that practitioners will also find it useful. It provides a clear description of the *concepts* that underlie operating systems. As prerequisites, we assume that the reader is familiar with basic data structures, computer organization, and a high-level language, such as C or Java. The hardware topics required for an understanding of operating systems are covered in Chapter 1. In that chapter, we also include an overview of the fundamental data structures that are prevalent in most operating systems. For code examples, we use predominantly C, as well as a significant amount of Java, but the reader can still understand the algorithms without a thorough knowledge of these languages.

Concepts are presented using intuitive descriptions. Important theoretical results are covered, but formal proofs are largely omitted. The bibliographical notes at the end of each chapter contain pointers to research papers in which results were first presented and proved, as well as references to recent material for further reading. In place of proofs, figures and examples are used to suggest why we should expect the result in question to be true.

The fundamental concepts and algorithms covered in the book are often based on those used in both open-source and commercial operating systems. Our aim is to present these concepts and algorithms in a general setting that is not tied to one particular operating system. However, we present a large number of examples that pertain to the most popular and the most innovative operating systems, including Linux, Microsoft Windows, Apple macOS (the original name, OS X, was changed in 2016 to match the naming scheme of other Apple products), and Solaris. We also include examples of both Android and iOS, currently the two dominant mobile operating systems.

The organization of the text reflects our many years of teaching courses on operating systems. Consideration was also given to the feedback provided by the reviewers of the text, along with the many comments and suggestions we received from readers of our previous editions and from our current and former students. This Tenth Edition also reflects most of the curriculum guidelines in the operating-systems area in *Computer Science Curricula 2013*, the most recent curriculum guidelines for undergraduate degree programs in computer science published by the IEEE Computing Society and the Association for Computing Machinery (ACM).

## What's New in This Edition

For the Tenth Edition, we focused on revisions and enhancements aimed at lowering costs to the students, better engaging them in the learning process, and providing increased support for instructors.

According to the publishing industry's most trusted market research firm, Outsell, 2015 represented a turning point in text usage: for the first time, student preference for digital learning materials was higher than for print, and the increase in preference for digital has been accelerating since.

While print remains important for many students as a pedagogical tool, the Tenth Edition is being delivered in forms that emphasize support for learning from digital materials. All forms we are providing dramatically reduce the cost to students compared to the Ninth Edition. These forms are:

- **Stand-alone e-text now with significant enhancements**. The e-text format for the Tenth Edition adds exercises with solutions at the ends of main sections, hide/reveal definitions for key terms, and a number of animated figures. It also includes additional "Practice Exercises" with solutions for each chapter, extra exercises, programming problems and projects, "Further Reading" sections, a complete glossary, and four appendices for legacy operating systems.

- **E-text with print companion bundle**. For a nominal additional cost, the e-text also is available with an abridged print companion that includes a loose-leaf copy of the main chapter text, end-of-chapter "Practice Exercises" (solutions available online), and "Further Reading" sections. Instructors may also order bound print companions for the bundled package by contacting their Wiley account representative.

Although we highly encourage all instructors and students to take advantage of the cost, content, and learning advantages of the e-text edition, it is possible for instructors to work with their Wiley Account Manager to create a custom print edition.

To explore these options further or to discuss other options, contact your Wiley account manager (http://www.wiley.com/go/whosmyrep) or visit the product information page for this text on wiley.com

## Book Material

The book consists of 21 chapters and 4 appendices. Each chapter and appendix contains the text, as well as the following enhancements:

- A set of practice exercises, including solutions
- A set of regular exercises
- A set of programming problems
- A set of programming projects
- A Further Reading section
- Pop-up definitions of important (blue) terms
- A glossary of important terms
- Animations that describe specific key concepts

A hard copy of the text is available in book stores and online. That version has the same text chapters as the electronic version. It does not, however, include the appendices, the regular exercises, the solutions to the practice exercises, the programming problems, the programming projects, and some of the other enhancements found in this ePub electronic book.

## Content of This Book

The text is organized in ten major parts:

- **Overview**. Chapters 1 and 2 explain what operating systems are, what they do, and how they are designed and constructed. These chapters discuss what the common features of an operating system are and what an operating system does for the user. We include coverage of both traditional PC and server operating systems and operating systems for mobile devices. The presentation is motivational and explanatory in nature. We have avoided a discussion of how things are done internally in these chapters. Therefore, they are suitable for individual readers or for students in lower-level classes who want to learn what an operating system is without getting into the details of the internal algorithms.

- **Process management**. Chapters 3 through 5 describe the process concept and concurrency as the heart of modern operating systems. A ***process*** is the unit of work in a system. Such a system consists of a collection of ***concurrently*** executing processes, some executing operating-system code and others executing user code. These chapters cover methods for process scheduling and interprocess communication. Also included is a detailed discussion of threads, as well as an examination of issues related to multicore systems and parallel programming.

- **Process synchronization**. Chapters 6 through 8 cover methods for process synchronization and deadlock handling. Because we have increased the coverage of process synchronization, we have divided the former Chapter 5 (Process Synchronization) into two separate chapters: Chapter 6, Synchronization Tools, and Chapter 7, Synchronization Examples.

- **Memory management**. Chapters 9 and 10 deal with the management of main memory during the execution of a process. To improve both the utilization of the CPU and the speed of its response to its users, the computer must keep several processes in memory. There are many different memory-management schemes, reflecting various approaches to memory management, and the effectiveness of a particular algorithm depends on the situation.

- **Storage management**. Chapters 11 and 12 describe how mass storage and I/O are handled in a modern computer system. The I/O devices that attach to a computer vary widely, and the operating system needs to provide a wide range of functionality to applications to allow them to control all aspects of these devices. We discuss system I/O in depth, including I/O system design, interfaces, and internal system structures and functions. In many ways, I/O devices are the slowest major components of the computer. Because they represent a performance bottleneck, we also examine performance issues associated with I/O devices.

- **File systems**. Chapters 13 through 15 discuss how file systems are handled in a modern computer system. File systems provide the mechanism for online storage of and access to both data and programs. We describe the classic internal algorithms and structures of storage management and provide a firm practical understanding of the algorithms used—their properties, advantages, and disadvantages.

- **Security and protection**. Chapters 16 and 17 discuss the mechanisms necessary for the security and protection of computer systems. The processes in an operating system must be protected from one another's activities. To provide such protection, we must ensure that only processes that have gained proper authorization from the operating system can operate on the files, memory, CPU, and other resources of the system. Protection is a mechanism for controlling the access of programs, processes, or users to computer-system resources. This mechanism must provide a means of specifying the controls to be imposed, as well as a means of enforcement. Security protects the integrity of the information stored in the system (both data and code), as well as the physical resources of the system, from unauthorized access, malicious destruction or alteration, and accidental introduction of inconsistency.

- **Advanced topics**. Chapters 18 and 19 discuss virtual machines and networks/distributed systems. Chapter 18 provides an overview of virtual machines and their relationship to contemporary operating systems. Included is a general description of the hardware and software techniques that make virtualization possible. Chapter 19 provides an overview of computer networks and distributed systems, with a focus on the Internet and TCP/IP.

- **Case studies**. Chapter 20 and 21 present detailed case studies of two real operating systems—Linux and Windows 10.

- **Appendices**. Appendix A discusses several old influential operating systems that are no longer in use. Appendices B through D cover in great detaisl three older operating systems— Windows 7, BSD, and Mach.

## Programming Environments

The text provides several example programs written in C and Java. These programs are intended to run in the following programming environments:

- **POSIX**. POSIX (which stands for *Portable Operating System Interface*) represents a set of standards implemented primarily for UNIX-based operating systems. Although Windows systems can also run certain POSIX programs, our coverage of POSIX focuses on Linux and UNIX systems. POSIX-compliant systems must implement the POSIX core standard (POSIX.1); Linux and macOS are examples of POSIX-compliant systems. POSIX also defines several extensions to the standards, including real-time extensions (POSIX.1b) and an extension for a threads library (POSIX.1c, better known as Pthreads). We provide several programming examples written in C illustrating the POSIX base API, as well as Pthreads and the extensions for real-time programming. These example programs were tested on Linux 4.4 and macOS 10.11 systems using the `gcc` compiler.
- **Java**. Java is a widely used programming language with a rich API and built-in language support for concurrent and parallel programming. Java programs run on any operating system supporting a Java virtual machine (or JVM). We illustrate various operating-system and networking concepts with Java programs tested using Version 1.8 of the Java Development Kit (JDK).
- **Windows systems**. The primary programming environment for Windows systems is the Windows API, which provides a comprehensive set of functions for managing processes, threads, memory, and peripheral devices. We supply a modest number of C programs illustrating the use of this API. Programs were tested on a system running Windows 10.

We have chosen these three programming environments because we believe that they best represent the two most popular operating-system models—Linux/UNIX and Windows—along with the widely used Java environment. Most programming examples are written in C, and we expect readers to be comfortable with this language. Readers familiar with both the C and Java languages should easily understand most programs provided in this text.

In some instances—such as thread creation—we illustrate a specific concept using all three programming environments, allowing the reader to contrast the three different libraries as they address the same task. In other situations, we may use just one of the APIs to demonstrate a concept. For example, we illustrate shared memory using just the POSIX API; socket programming in TCP/IP is highlighted using the Java API.

## Linux Virtual Machine

To help students gain a better understanding of the Linux system, we provide a Linux virtual machine running the Ubuntu distribution with this text. The virtual machine, which is available for download from the text website (http://www.os-book.com), also provides development environments including the gcc and Java compilers. Most of the programming assignments in the book can be completed using this virtual machine, with the exception of assignments that require the Windows API. The virtual machine can be installed and run on any host operating system that can run the VirtualBox virtualization software, which currently includes Windows 10 Linux, and macOS.

## The Tenth Edition

As we wrote this Tenth Edition of *Operating System Concepts,* we were guided by the sustained growth in four fundamental areas that affect operating systems:

1. Mobile operating systems
2. Multicore systems
3. Virtualization
4. Nonvolatile memory secondary storage

To emphasize these topics, we have integrated relevant coverage throughout this new edition. For example, we have greatly increased our coverage of the Android and iOS mobile operating systems, as well as our coverage of the ARMv8 architecture that dominates mobile devices. We have also increased our coverage of multicore systems, including increased coverage of APIs that provide support for concurrency and parallelism. Nonvolatile memory devices like SSDs are now treated as the equals of hard-disk drives in the chapters that discuss I/O, mass storage, and file systems.

Several of our readers have expressed support for an increase in Java coverage, and we have provided additional Java examples throughout this edition.

Additionally, we have rewritten material in almost every chapter by bringing older material up to date and removing material that is no longer interesting or relevant. We have reordered many chapters and have, in some instances, moved sections from one chapter to another. We have also greatly revised the artwork, creating several new figures as well as modifying many existing figures.

### Major Changes

The Tenth Edition update encompasses much more material than previous updates, in terms of both content and new supporting material. Next, we provide a brief outline of the major content changes in each chapter:

- **Chapter 1: Introduction** includes updated coverage of multicore systems, as well as new coverage of NUMA systems and Hadoop clusters. Old material has been updated, and new motivation has been added for the study of operating systems.
- **Chapter 2: Operating-System Structures** provides a significantly revised discussion of the design and implementation of operating systems. We have updated our treatment of Android and iOS and have revised our

coverage of the system boot process with a focus on GRUB for Linux systems. New coverage of the Windows subsystem for Linux is included as well. We have added new sections on linkers and loaders, and we now discuss why applications are often operating-system specific. Finally, we have added a discussion of the BCC debugging toolset.

- **Chapter 3: Processes** simplifies the discussion of scheduling so that it now includes only CPU scheduling issues. New coverage describes the memory layout of a C program, the Android process hierarchy, Mach message passing, and Android RPCs. We have also replaced coverage of the traditional UNIX/Linux `init` process with coverage of `systemd`.
- **Chapter 4: Threads and Concurrency** (previously Threads) increases the coverage of support for concurrent and parallel programming at the API and library level. We have revised the section on Java threads so that it now includes futures and have updated the coverage of Apple's Grand Central Dispatch so that it now includes Swift. New sections discuss fork-join parallelism using the fork-join framework in Java, as well as Intel thread building blocks.
- **Chapter 5: CPU Scheduling** (previously Chapter 6) revises the coverage of multilevel queue and multicore processing scheduling. We have integrated coverage of NUMA-aware scheduling issues throughout, including how this scheduling affects load balancing. We also discuss related modifications to the Linux CFS scheduler. New coverage combines discussions of round-robin and priority scheduling, heterogeneous multiprocessing, and Windows 10 scheduling.
- **Chapter 6: Synchronization Tools** (previously part of Chapter 5, Process Synchronization) focuses on various tools for synchronizing processes. Significant new coverage discusses architectural issues such as instruction reordering and delayed writes to buffers. The chapter also introduces lock-free algorithms using compare-and-swap (CAS) instructions. No specific APIs are presented; rather, the chapter provides an introduction to race conditions and general tools that can be used to prevent data races. Details include new coverage of memory models, memory barriers, and liveness issues.
- **Chapter 7: Synchronization Examples** (previously part of Chapter 5, Process Synchronization) introduces classical synchronization problems and discusses specific API support for designing solutions that solve these problems. The chapter includes new coverage of POSIX named and unnamed semaphores, as well as condition variables. A new section on Java synchronization is included as well.
- **Chapter 8: Deadlocks** (previously Chapter 7) provides minor updates, including a new section on livelock and a discussion of deadlock as an example of a liveness hazard. The chapter includes new coverage of the Linux `lockdep` and the BCC `deadlock_detector` tools, as well as coverage of Java deadlock detection using thread dumps.
- **Chapter 9: Main Memory** (previously Chapter 8) includes several revisions that bring the chapter up to date with respect to memory manage-

ment on modern computer systems. We have added new coverage of the ARMv8 64-bit architecture, updated the coverage of dynamic link libraries, and changed swapping coverage so that it now focuses on swapping pages rather than processes. We have also eliminated coverage of segmentation.

- **Chapter 10: Virtual Memory** (previously Chapter 9) contains several revisions, including updated coverage of memory allocation on NUMA systems and global allocation using a free-frame list. New coverage includes compressed memory, major/minor page faults, and memory management in Linux and Windows 10.
- **Chapter 11: Mass-Storage Structure** (previously Chapter 10) adds coverage of nonvolatile memory devices, such as flash and solid-state disks. Hard-drive scheduling is simplified to show only currently used algorithms. Also included are a new section on cloud storage, updated RAID coverage, and a new discussion of object storage.
- **Chapter 12, I/O** (previously Chapter 13) updates the coverage of technologies and performance numbers, expands the coverage of synchronous/asynchronous and blocking/nonblocking I/O, and adds a section on vectored I/O. It also expands coverage of power management for mobile operating systems.
- **Chapter 13: File-System Interface** (previously Chapter 11) has been updated with information about current technologies. In particular, the coverage of directory structures has been improved, and the coverage of protection has been updated. The memory-mapped files section has been expanded, and a Windows API example has been added to the discussion of shared memory. The ordering of topics is refactored in Chapter 13 and 14.
- **Chapter 14: File-System Implementation** (previously Chapter 12) has been updated with coverage of current technologies. The chapter now includes discussions of TRIM and the Apple File System. In addition, the discussion of performance has been updated, and the coverage of journaling has been expanded.
- **Chapter 15: File System Internals** is new and contains updated information from previous Chapters 11 and 12.
- **Chapter 16: Security** (previously Chapter 15) now precedes the protection chapter. It includes revised and updated terms for current security threats and solutions, including ransomware and remote access tools. The principle of least privilege is emphasized. Coverage of code-injection vulnerabilities and attacks has been revised and now includes code samples. Discussion of encryption technologies has been updated to focus on the technologies currently used. Coverage of authentication (by passwords and other methods) has been updated and expanded with helpful hints. Additions include a discussion of address-space layout randomization and a new summary of security defenses. The Windows 7 example has been updated to Windows 10.
- **Chapter 17: Protection** (previously Chapter 14) contains major changes. The discussion of protection rings and layers has been updated and now

refers to the Bell–LaPadula model and explores the ARM model of TrustZones and Secure Monitor Calls. Coverage of the need-to-know principle has been expanded, as has coverage of mandatory access control. Subsections on Linux capabilities, Darwin entitlements, security integrity protection, system-call filtering, sandboxing, and code signing have been added. Coverage of run-time-based enforcement in Java has also been added, including the stack inspection technique.

- **Chapter 18: Virtual Machines** (previously Chapter 16) includes added details about hardware assistance technologies. Also expanded is the topic of application containment, now including containers, zones, docker, and Kubernetes. A new section discusses ongoing virtualization research, including unikernels, library operating systems, partitioning hypervisors, and separation hypervisors.
- **Chapter 19, Networks and Distributed Systems** (previously Chapter 17) has been substantially updated and now combines coverage of computer networks and distributed systems. The material has been revised to bring it up to date with respect to contemporary computer networks and distributed systems. The TCP/IP model receives added emphasis, and a discussion of cloud storage has been added. The section on network topologies has been removed. Coverage of name resolution has been expanded and a Java example added. The chapter also includes new coverage of distributed file systems, including MapReduce on top of Google file system, Hadoop, GPFS, and Lustre.
- **Chapter 20: The Linux System** (previously Chapter 18) has been updated to cover the Linux 4.*i* kernel.
- **Chapter 21: The Windows 10 System** is a new chapter that covers the internals of Windows 10.
- **Appendix A: Influential Operating Systems** has been updated to include material from chapters that are no longer covered in the text.

## Supporting Website

When you visit the website supporting this text at http://www.os-book.com, you can download the following resources:

- Linux virtual machine
- C and Java source code
- The complete set of figures and illustrations
- FreeBSD, Mach, and Windows 7 case studies
- Errata
- Bibliography

## Notes to Instructors

On the website for this text, we provide several sample syllabi that suggest various approaches for using the text in both introductory and advanced courses.

As a general rule, we encourage instructors to progress sequentially through the chapters, as this strategy provides the most thorough study of operating systems. However, by using the sample syllabi, an instructor can select a different ordering of chapters (or subsections of chapters).

In this edition, we have added many new written exercises and programming problems and projects. Most of the new programming assignments involve processes, threads, process scheduling, process synchronization, and memory management. Some involve adding kernel modules to the Linux system, which requires using either the Linux virtual machine that accompanies this text or another suitable Linux distribution.

Solutions to written exercises and programming assignments are available to instructors who have adopted this text for their operating-system class. To obtain these restricted supplements, contact your local John Wiley & Sons sales representative. You can find your Wiley representative by going to http://www.wiley.com/college and clicking “Who’s my rep?”

## Notes to Students

We encourage you to take advantage of the practice exercises that appear at the end of each chapter. We also encourage you to read through the study guide, which was prepared by one of our students. Finally, for students who are unfamiliar with UNIX and Linux systems, we recommend that you download and install the Linux virtual machine that we include on the supporting website. Not only will this provide you with a new computing experience, but the open-source nature of Linux will allow you to easily examine the inner details of this popular operating system. We wish you the very best of luck in your study of operating systems!

## Contacting Us

We have endeavored to eliminate typos, bugs, and the like from the text. But, as in new releases of software, bugs almost surely remain. An up-to-date errata list is accessible from the book’s website. We would be grateful if you would notify us of any errors or omissions in the book that are not on the current list of errata.

We would be glad to receive suggestions on improvements to the book. We also welcome any contributions to the book website that could be of use to other readers, such as programming exercises, project suggestions, on-line labs and tutorials, and teaching tips. E-mail should be addressed to os-book-authors@cs.yale.edu.

## Acknowledgments

Many people have helped us with this Tenth Edition, as well as with the previous nine editions from which it is derived.

## Tenth Edition

- Rick Farrow provided expert advice as a technical editor.
- Jonathan Levin helped out with coverage of mobile systems, protection, and security.
- Alex Ionescu updated the previous Windows 7 chapter to provide Chapter 21: Windows 10.
- Sarah Diesburg revised Chapter 19: Networks and Distributed Systems.
- Brendan Gregg provided guidance on the BCC toolset.
- Richard Stallman (RMS) supplied feedback on the description of free and open-source software.
- Robert Love provided updates to Chapter 20: The Linux System.
- Michael Shapiro helped with storage and I/O technology details.
- Richard West provided insight on areas of virtualization research.
- Clay Breshears helped with coverage of Intel thread-building blocks.
- Gerry Howser gave feedback on motivating the study of operating systems and also tried out new material in his class.
- Judi Paige helped with generating figures and presentation of slides.
- Jay Gagne and Audra Rissmeyer prepared new artwork for this edition.
- Owen Galvin provided technical editing for Chapter 11 and Chapter 12.
- Mark Wogahn has made sure that the software to produce this book ($\LaTeX$ and fonts) works properly.
- Ranjan Kumar Meher rewrote some of the $\LaTeX$ software used in the production of this new text.

## Previous Editions

- **First three editions**. This book is derived from the previous editions, the first three of which were coauthored by James Peterson.
- **General contributions**. Others who helped us with previous editions include Hamid Arabnia, Rida Bazzi, Randy Bentson, David Black, Joseph Boykin, Jeff Brumfield, Gael Buckley, Roy Campbell, P. C. Capon, John Carpenter, Gil Carrick, Thomas Casavant, Bart Childs, Ajoy Kumar Datta, Joe Deck, Sudarshan K. Dhall, Thomas Doeppner, Caleb Drake, M. Rasit Eskicioğlu, Hans Flack, Robert Fowler, G. Scott Graham, Richard Guy, Max Hailperin, Rebecca Hartman, Wayne Hathaway, Christopher Haynes, Don Heller, Bruce Hillyer, Mark Holliday, Dean Hougen, Michael Huang, Ahmed Kamel, Morty Kewstel, Richard Kieburtz, Carol Kroll, Morty Kwestel, Thomas LeBlanc, John Leggett, Jerrold Leichter, Ted Leung, Gary Lippman, Carolyn Miller, Michael Molloy, Euripides Montagne, Yoichi Muraoka, Jim M. Ng, Banu Özden, Ed Posnak, Boris Putanec, Charles

Qualline, John Quarterman, Mike Reiter, Gustavo Rodriguez-Rivera, Carolyn J. C. Schauble, Thomas P. Skinner, Yannis Smaragdakis, Jesse St. Laurent, John Stankovic, Adam Stauffer, Steven Stepanek, John Sterling, Hal Stern, Louis Stevens, Pete Thomas, David Umbaugh, Steve Vinoski, Tommy Wagner, Larry L. Wear, John Werth, James M. Westall, J. S. Weston, and Yang Xiang

- **Specifi Contributions**
    - Robert Love updated both Chapter 20 and the Linux coverage throughout the text, as well as answering many of our Android-related questions.
    - Appendix B was written by Dave Probert and was derived from Chapter 22 of the Eighth Edition of *Operating System Concepts*.
    - Jonathan Katz contributed to Chapter 16. Richard West provided input into Chapter 18. Salahuddin Khan updated Section 16.7 to provide new coverage of Windows 7 security.
    - Parts of Chapter 19 were derived from a paper by Levy and Silberschatz [1990].
    - Chapter 20 was derived from an unpublished manuscript by Stephen Tweedie.
    - Cliff Martin helped with updating the UNIX appendix to cover FreeBSD.
    - Some of the exercises and accompanying solutions were supplied by Arvind Krishnamurthy.
    - Andrew DeNicola prepared the student study guide that is available on our website. Some of the slides were prepared by Marilyn Turnamian.
    - Mike Shapiro, Bryan Cantrill, and Jim Mauro answered several Solaris-related questions, and Bryan Cantrill from Sun Microsystems helped with the ZFS coverage. Josh Dees and Rob Reynolds contributed coverage of Microsoft's NET.
    - Owen Galvin helped copy-edit Chapter 18 edition.

## Book Production

The Executive Editor was Don Fowley. The Senior Production Editor was Ken Santor. The Freelance Developmental Editor was Chris Nelson. The Assistant Developmental Editor was Ryann Dannelly. The cover designer was Tom Nery. The copyeditor was Beverly Peavler. The freelance proofreader was Katrina Avery. The freelance indexer was WordCo, Inc. The Aptara LaTex team consisted of Neeraj Saxena and Lav kush.

## Personal Notes

Avi would like to acknowledge Valerie for her love, patience, and support during the revision of this book.

Peter would like to thank his wife Carla and his children, Gwen, Owen, and Maddie.

Greg would like to acknowledge the continued support of his family: his wife Pat and sons Thomas and Jay.

Abraham Silberschatz, New Haven, CT
Peter Baer Galvin, Boston, MA
Greg Gagne, Salt Lake City, UT

# Contents

## PART ONE ■ OVERVIEW

### Chapter 1 Introduction

1.1 What Operating Systems Do 4
1.2 Computer-System Organization 7
1.3 Computer-System Architecture 15
1.4 Operating-System Operations 21
1.5 Resource Management 27
1.6 Security and Protection 33
1.7 Virtualization 34
1.8 Distributed Systems 35
1.9 Kernel Data Structures 36
1.10 Computing Environments 40
1.11 Free and Open-Source Operating Systems 46
Practice Exercises 53
Further Reading 54

### Chapter 2 Operating-System Structures

2.1 Operating-System Services 55
2.2 User and Operating-System Interface 58
2.3 System Calls 62
2.4 System Services 74
2.5 Linkers and Loaders 75
2.6 Why Applications Are Operating-System Specific 77
2.7 Operating-System Design and Implementation 79
2.8 Operating-System Structure 81
2.9 Building and Booting an Operating System 92
2.10 Operating-System Debugging 95
2.11 Summary 100
Practice Exercises 101
Further Reading 101

## PART TWO ■ PROCESS MANAGEMENT

### Chapter 3 Processes

3.1 Process Concept 106
3.2 Process Scheduling 110
3.3 Operations on Processes 116
3.4 Interprocess Communication 123
3.5 IPC in Shared-Memory Systems 125
3.6 IPC in Message-Passing Systems 127
3.7 Examples of IPC Systems 132
3.8 Communication in Client–Server Systems 145
3.9 Summary 153
Practice Exercises 154
Further Reading 156

## Chapter 4 Threads & Concurrency

4.1 Overview 160
4.2 Multicore Programming 162
4.3 Multithreading Models 166
4.4 Thread Libraries 168
4.5 Implicit Threading 176
4.6 Threading Issues 188
4.7 Operating-System Examples 194
4.8 Summary 196
Practice Exercises 197
Further Reading 198

## Chapter 5 CPU Scheduling

5.1 Basic Concepts 200
5.2 Scheduling Criteria 204
5.3 Scheduling Algorithms 205
5.4 Thread Scheduling 217
5.5 Multi-Processor Scheduling 220
5.6 Real-Time CPU Scheduling 227
5.7 Operating-System Examples 234
5.8 Algorithm Evaluation 244
5.9 Summary 250
Practice Exercises 251
Further Reading 254

# PART THREE ■ PROCESS SYNCHRONIZATION

## Chapter 6 Synchronization Tools

6.1 Background 257
6.2 The Critical-Section Problem 260
6.3 Peterson's Solution 262
6.4 Hardware Support for Synchronization 265
6.5 Mutex Locks 270
6.6 Semaphores 272
6.7 Monitors 276
6.8 Liveness 283
6.9 Evaluation 284
6.10 Summary 286
Practice Exercises 287
Further Reading 288

## Chapter 7 Synchronization Examples

7.1 Classic Problems of Synchronization 289
7.2 Synchronization within the Kernel 295
7.3 POSIX Synchronization 299
7.4 Synchronization in Java 303
7.5 Alternative Approaches 311
7.6 Summary 314
Practice Exercises 314
Further Reading 315

## Chapter 8 Deadlocks

8.1 System Model 318
8.2 Deadlock in Multithreaded Applications 319
8.3 Deadlock Characterization 321
8.4 Methods for Handling Deadlocks 326
8.5 Deadlock Prevention 327
8.6 Deadlock Avoidance 330
8.7 Deadlock Detection 337
8.8 Recovery from Deadlock 341
8.9 Summary 343
Practice Exercises 344
Further Reading 346

# PART FOUR ■ MEMORY MANAGEMENT

## Chapter 9 Main Memory

9.1 Background 349
9.2 Contiguous Memory Allocation 356
9.3 Paging 360
9.4 Structure of the Page Table 371
9.5 Swapping 376
9.6 Example: Intel 32- and 64-bit Architectures 379
9.7 Example: ARMv8 Architecture 383
9.8 Summary 384
Practice Exercises 385
Further Reading 387

## Chapter 10 Virtual Memory

10.1 Background 389
10.2 Demand Paging 392
10.3 Copy-on-Write 399
10.4 Page Replacement 401
10.5 Allocation of Frames 413
10.6 Thrashing 419
10.7 Memory Compression 425
10.8 Allocating Kernel Memory 426
10.9 Other Considerations 430
10.10 Operating-System Examples 436
10.11 Summary 440
Practice Exercises 441
Further Reading 444

# PART FIVE ■ STORAGE MANAGEMENT

## Chapter 11 Mass-Storage Structure

11.1 Overview of Mass-Storage Structure 449
11.2 HDD Scheduling 457
11.3 NVM Scheduling 461
11.4 Error Detection and Correction 462
11.5 Storage Device Management 463
11.6 Swap-Space Management 467
11.7 Storage Attachment 469
11.8 RAID Structure 473
11.9 Summary 485
Practice Exercises 486
Further Reading 487

## Chapter 12 I/O Systems

12.1 Overview 489
12.2 I/O Hardware 490
12.3 Application I/O Interface 500
12.4 Kernel I/O Subsystem 508
12.5 Transforming I/O Requests to Hardware Operations 516
12.6 STREAMS 519
12.7 Performance 521
12.8 Summary 524
Practice Exercises 525
Further Reading 526

# PART SIX ■ FILE SYSTEM

## Chapter 13 File-System Interface

13.1 File Concept 529
13.2 Access Methods 539
13.3 Directory Structure 541
13.4 Protection 550
13.5 Memory-Mapped Files 555
13.6 Summary 560
Practice Exercises 560
Further Reading 561

## Chapter 14 File-System Implementation

14.1 File-System Structure 564
14.2 File-System Operations 566
14.3 Directory Implementation 568
14.4 Allocation Methods 570
14.5 Free-Space Management 578
14.6 Efficiency and Performance 582
14.7 Recovery 586
14.8 Example: The WAFL File System 589
14.9 Summary 593
Practice Exercises 594
Further Reading 594

## Chapter 15 File-System Internals

15.1 File Systems 597
15.2 File-System Mounting 598
15.3 Partitions and Mounting 601
15.4 File Sharing 602
15.5 Virtual File Systems 603
15.6 Remote File Systems 605
15.7 Consistency Semantics 608
15.8 NFS 610
15.9 Summary 615
Practice Exercises 616
Further Reading 617

# PART SEVEN ■ SECURITY AND PROTECTION

## Chapter 16 Security

16.1 The Security Problem 621
16.2 Program Threats 625
16.3 System and Network Threats 634
16.4 Cryptography as a Security Tool 637
16.5 User Authentication 648
16.6 Implementing Security Defenses 653
16.7 An Example: Windows 10 662
16.8 Summary 664
Further Reading 665

## Chapter 17 Protection

17.1 Goals of Protection 667
17.2 Principles of Protection 668
17.3 Protection Rings 669
17.4 Domain of Protection 671
17.5 Access Matrix 675
17.6 Implementation of the Access Matrix 679
17.7 Revocation of Access Rights 682
17.8 Role-Based Access Control 683
17.9 Mandatory Access Control (MAC) 684
17.10 Capability-Based Systems 685
17.11 Other Protection Improvement Methods 687
17.12 Language-Based Protection 690
17.13 Summary 696
Further Reading 697

# PART EIGHT ■ ADVANCED TOPICS

## Chapter 18 Virtual Machines

18.1 Overview 701
18.2 History 703
18.3 Benefits and Features 704
18.4 Building Blocks 707
18.5 Types of VMs and Their Implementations 713
18.6 Virtualization and Operating-System Components 719
18.7 Examples 726
18.8 Virtualization Research 728
18.9 Summary 729
Further Reading 730

## Chapter 19 Networks and Distributed Systems

19.1 Advantages of Distributed Systems 733
19.2 Network Structure 735
19.3 Communication Structure 738
19.4 Network and Distributed Operating Systems 749
19.5 Design Issues in Distributed Systems 753
19.6 Distributed File Systems 757
19.7 DFS Naming and Transparency 761
19.8 Remote File Access 764
19.9 Final Thoughts on Distributed File Systems 767
19.10 Summary 768
Practice Exercises 769
Further Reading 770

# PART NINE ■ CASE STUDIES

## Chapter 20 The Linux System

20.1 Linux History 775
20.2 Design Principles 780
20.3 Kernel Modules 783
20.4 Process Management 786
20.5 Scheduling 790
20.6 Memory Management 795
20.7 File Systems 803
20.8 Input and Output 810
20.9 Interprocess Communication 812
20.10 Network Structure 813
20.11 Security 816
20.12 Summary 818
Practice Exercises 819
Further Reading 819

## Chapter 21 Windows 10

21.1 History 821
21.2 Design Principles 826
21.3 System Components 838
21.4 Terminal Services and Fast User Switching 874
21.5 File System 875
21.6 Networking 880
21.7 Programmer Interface 884
21.8 Summary 895
Practice Exercises 896
Further Reading 897

# PART TEN ■ APPENDICES

## Chapter A Influential Operating Systems

A.1 Feature Migration 1
A.2 Early Systems 2
A.3 Atlas 9
A.4 XDS-940 10
A.5 THE 11
A.6 RC 4000 11
A.7 CTSS 12
A.8 MULTICS 13
A.9 IBM OS/360 13
A.10 TOPS-20 15
A.11 CP/M and MS/DOS 15
A.12 Macintosh Operating System and Windows 16
A.13 Mach 16
A.14 Capability-based Systems—Hydra and CAP 18
A.15 Other Systems 20
Further Reading 21

## Chapter B Windows 7

B.1 History 1
B.2 Design Principles 3
B.3 System Components 10
B.4 Terminal Services and Fast User Switching 34
B.5 File System 35
B.6 Networking 41
B.7 Programmer Interface 46
B.8 Summary 55
Practice Exercises 55
Further Reading 56

## Chapter C BSD UNIX

C.1 UNIX History 1
C.2 Design Principles 6
C.3 Programmer Interface 8
C.4 User Interface 15
C.5 Process Management 18
C.6 Memory Management 22
C.7 File System 25
C.8 I/O System 33
C.9 Interprocess Communication 36
C.10 Summary 41
Further Reading 42

## Chapter D The Mach System

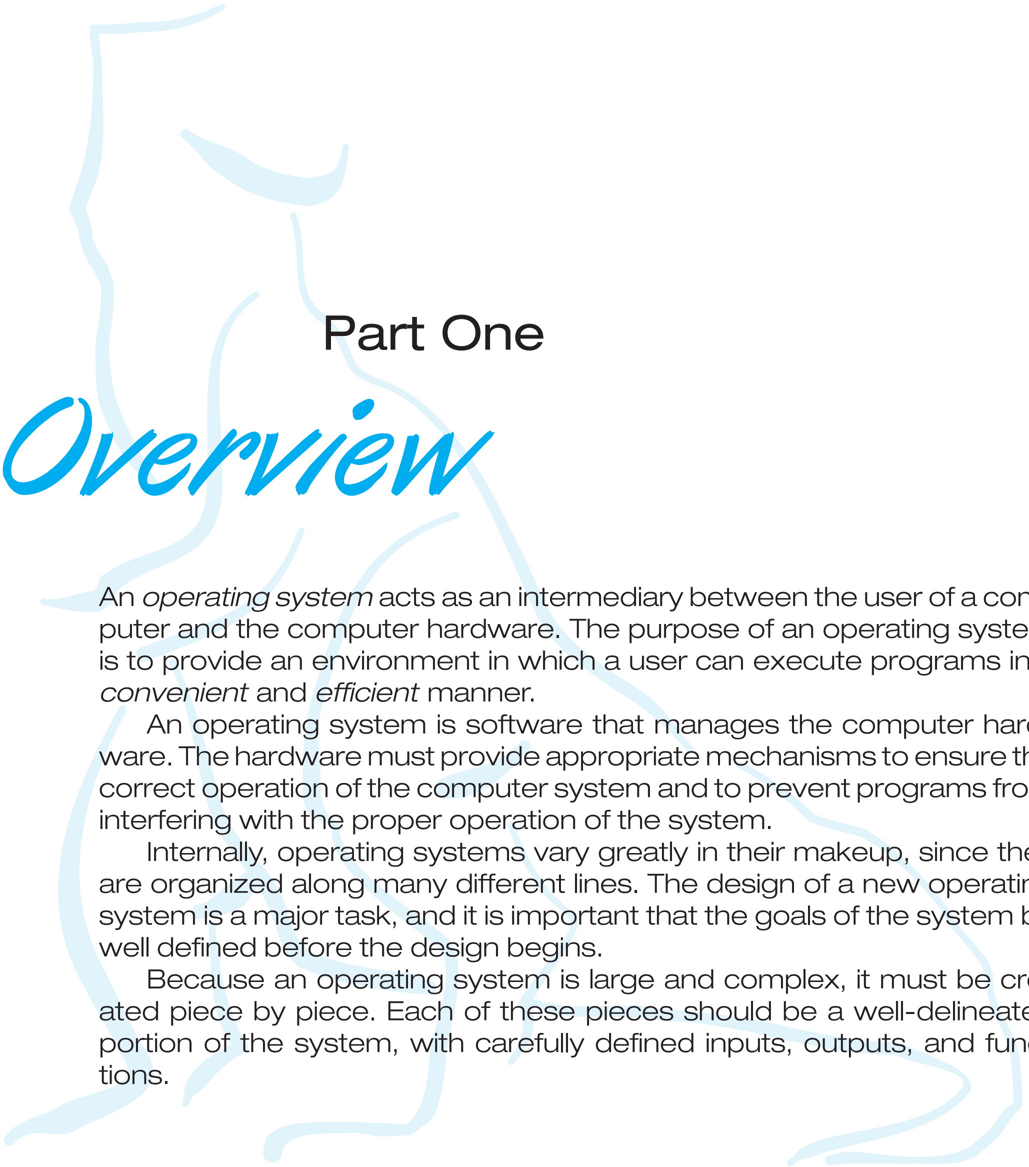
D.1 History of the Mach System 1
D.2 Design Principles 3
D.3 System Components 4
D.4 Process Management 7
D.5 Interprocess Communication 13
D.6 Memory Management 18
D.7 Programmer Interface 23
D.8 Summary 24
Further Reading 25

**Credits 963**

**Index 965**

# Part One

# Overview



An *operating system* acts as an intermediary between the user of a computer and the computer hardware. The purpose of an operating system is to provide an environment in which a user can execute programs in a *convenient* and *efficient* manner.

An operating system is software that manages the computer hardware. The hardware must provide appropriate mechanisms to ensure the correct operation of the computer system and to prevent programs from interfering with the proper operation of the system.

Internally, operating systems vary greatly in their makeup, since they are organized along many different lines. The design of a new operating system is a major task, and it is important that the goals of the system be well defined before the design begins.

Because an operating system is large and complex, it must be created piece by piece. Each of these pieces should be a well-delineated portion of the system, with carefully defined inputs, outputs, and functions.

# CHAPTER 1

# Introduction

An **operating system** is software that manages a computer's hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware. An amazing aspect of operating systems is how they vary in accomplishing these tasks in a wide variety of computing environments. Operating systems are everywhere, from cars and home appliances that include "Internet of Things" devices, to smart phones, personal computers, enterprise computers, and cloud computing environments.

In order to explore the role of an operating system in a modern computing environment, it is important first to understand the organization and architecture of computer hardware. This includes the CPU, memory, and I/O devices, as well as storage. A fundamental responsibility of an operating system is to allocate these resources to programs.

Because an operating system is large and complex, it must be created piece by piece. Each of these pieces should be a well-delineated portion of the system, with carefully defined inputs, outputs, and functions. In this chapter, we provide a general overview of the major components of a contemporary computer system as well as the functions provided by the operating system. Additionally, we cover several topics to help set the stage for the remainder of the text: data structures used in operating systems, computing environments, and open-source and free operating systems.

## CHAPTER OBJECTIVES

- Describe the general organization of a computer system and the role of interrupts.
- Describe the components in a modern multiprocessor computer system.
- Illustrate the transition from user mode to kernel mode.
- Discuss how operating systems are used in various computing environments.
- Provide examples of free and open-source operating systems.

## 1.1 What Operating Systems Do

We begin our discussion by looking at the operating system's role in the overall computer system. A computer system can be divided roughly into four components: the ***hardware***, the ***operating system***, the ***application programs***, and a ***user*** (Figure 1.1).

The **hardware**—the central processing unit (CPU), the memory, and the input/output (I/O) devices—provides the basic computing resources for the system. The **application programs**—such as word processors, spreadsheets, compilers, and web browsers—define the ways in which these resources are used to solve users' computing problems. The operating system controls the hardware and coordinates its use among the various application programs for the various users.

We can also view a computer system as consisting of hardware, software, and data. The operating system provides the means for proper use of these resources in the operation of the computer system. An operating system is similar to a government. Like a government, it performs no useful function by itself. It simply provides an ***environment*** within which other programs can do useful work.

To understand more fully the operating system's role, we next explore operating systems from two viewpoints: that of the user and that of the system.

### 1.1.1 User View

The user's view of the computer varies according to the interface being used. Many computer users sit with a laptop or in front of a PC consisting of a monitor, keyboard, and mouse. Such a system is designed for one user to monopolize its resources. The goal is to maximize the work (or play) that the user is performing. In this case, the operating system is designed mostly for **ease of use**, with some attention paid to performance and security and none paid to **resource utilization**—how various hardware and software resources are shared.

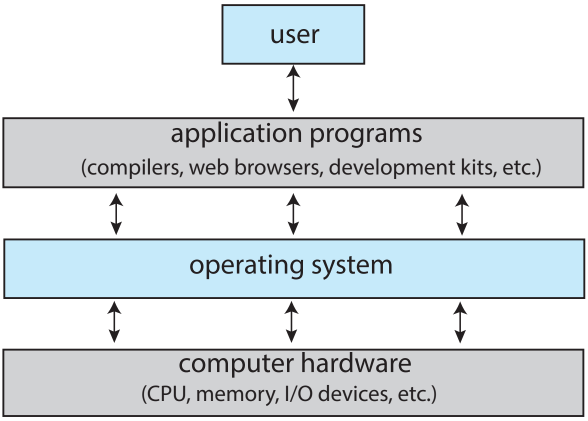

user
application programs
(compilers, web browsers, development kits, etc.)
operating system
computer hardware
(CPU, memory, I/O devices, etc.)

**Figure 1.1** Abstract view of the components of a computer system.

Increasingly, many users interact with mobile devices such as smartphones and tablets—devices that are replacing desktop and laptop computer systems for some users. These devices are typically connected to networks through cellular or other wireless technologies. The user interface for mobile computers generally features a **touch screen**, where the user interacts with the system by pressing and swiping fingers across the screen rather than using a physical keyboard and mouse. Many mobile devices also allow users to interact through a **voice recognition** interface, such as Apple's **Siri**.

Some computers have little or no user view. For example, **embedded computers** in home devices and automobiles may have numeric keypads and may turn indicator lights on or off to show status, but they and their operating systems and applications are designed primarily to run without user intervention.

### 1.1.2 System View

From the computer's point of view, the operating system is the program most intimately involved with the hardware. In this context, we can view an operating system as a **resource allocator**. A computer system has many resources that may be required to solve a problem: CPU time, memory space, storage space, I/O devices, and so on. The operating system acts as the manager of these resources. Facing numerous and possibly conflicting requests for resources, the operating system must decide how to allocate them to specific programs and users so that it can operate the computer system efficiently and fairly.

A slightly different view of an operating system emphasizes the need to control the various I/O devices and user programs. An operating system is a control program. A **control program** manages the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices.

### 1.1.3 Defining Operating Systems

By now, you can probably see that the term ***operating system*** covers many roles and functions. That is the case, at least in part, because of the myriad designs and uses of computers. Computers are present within toasters, cars, ships, spacecraft, homes, and businesses. They are the basis for game machines, cable TV tuners, and industrial control systems.

To explain this diversity, we can turn to the history of computers. Although computers have a relatively short history, they have evolved rapidly. Computing started as an experiment to determine what could be done and quickly moved to fixed-purpose systems for military uses, such as code breaking and trajectory plotting, and governmental uses, such as census calculation. Those early computers evolved into general-purpose, multifunction mainframes, and that's when operating systems were born. In the 1960s, **Moore's Law** predicted that the number of transistors on an integrated circuit would double every 18 months, and that prediction has held true. Computers gained in functionality and shrank in size, leading to a vast number of uses and a vast number and variety of operating systems. (See Appendix A for more details on the history of operating systems.)

How, then, can we define what an operating system is? In general, we have no completely adequate definition of an operating system. Operating systems exist because they offer a reasonable way to solve the problem of creating a usable computing system. The fundamental goal of computer systems is to execute programs and to make solving user problems easier. Computer hardware is constructed toward this goal. Since bare hardware alone is not particularly easy to use, application programs are developed. These programs require certain common operations, such as those controlling the I/O devices. The common functions of controlling and allocating resources are then brought together into one piece of software: the operating system.

In addition, we have no universally accepted definition of what is part of the operating system. A simple viewpoint is that it includes everything a vendor ships when you order "the operating system." The features included, however, vary greatly across systems. Some systems take up less than a megabyte of space and lack even a full-screen editor, whereas others require gigabytes of space and are based entirely on graphical windowing systems. A more common definition, and the one that we usually follow, is that the operating system is the one program running at all times on the computer—usually called the **kernel**. Along with the kernel, there are two other types of programs: **system programs**, which are associated with the operating system but are not necessarily part of the kernel, and application programs, which include all programs not associated with the operation of the system.

The matter of what constitutes an operating system became increasingly important as personal computers became more widespread and operating systems grew increasingly sophisticated. In 1998, the United States Department of Justice filed suit against Microsoft, in essence claiming that Microsoft included too much functionality in its operating systems and thus prevented application vendors from competing. (For example, a web browser was an integral part of Microsoft's operating systems.) As a result, Microsoft was found guilty of using its operating-system monopoly to limit competition.

Today, however, if we look at operating systems for mobile devices, we see that once again the number of features constituting the operating system is increasing. Mobile operating systems often include not only a core kernel but also **middleware**—a set of software frameworks that provide additional services to application developers. For example, each of the two most prominent mobile operating systems—Apple's iOS and Google's Android—features

### *WHY STUDY OPERATING SYSTEMS?*

Although there are many practitioners of computer science, only a small percentage of them will be involved in the creation or modification of an operating system. Why, then, study operating systems and how they work? Simply because, as almost all code runs on top of an operating system, knowledge of how operating systems work is crucial to proper, efficient, effective, and secure programming. Understanding the fundamentals of operating systems, how they drive computer hardware, and what they provide to applications is not only essential to those who program them but also highly useful to those who write programs on them and use them.

a core kernel along with middleware that supports databases, multimedia, and graphics (to name only a few).

In summary, for our purposes, the operating system includes the always-running kernel, middleware frameworks that ease application development and provide features, and system programs that aid in managing the system while it is running. Most of this text is concerned with the kernel of general-purpose operating systems, but other components are discussed as needed to fully explain operating system design and operation.

## 1.2 Computer-System Organization

A modern general-purpose computer system consists of one or more CPUs and a number of device controllers connected through a common **bus** that provides access between components and shared memory (Figure 1.2). Each device controller is in charge of a specific type of device (for example, a disk drive, audio device, or graphics display). Depending on the controller, more than one device may be attached. For instance, one system USB port can connect to a USB hub, to which several devices can connect. A device controller maintains some local buffer storage and a set of special-purpose registers. The device controller is responsible for moving the data between the peripheral devices that it controls and its local buffer storage.

Typically, operating systems have a **device driver** for each device controller. This device driver understands the device controller and provides the rest of the operating system with a uniform interface to the device. The CPU and the device controllers can execute in parallel, competing for memory cycles. To ensure orderly access to the shared memory, a memory controller synchronizes access to the memory.

In the following subsections, we describe some basics of how such a system operates, focusing on three key aspects of the system. We start with interrupts, which alert the CPU to events that require attention. We then discuss storage structure and I/O structure.

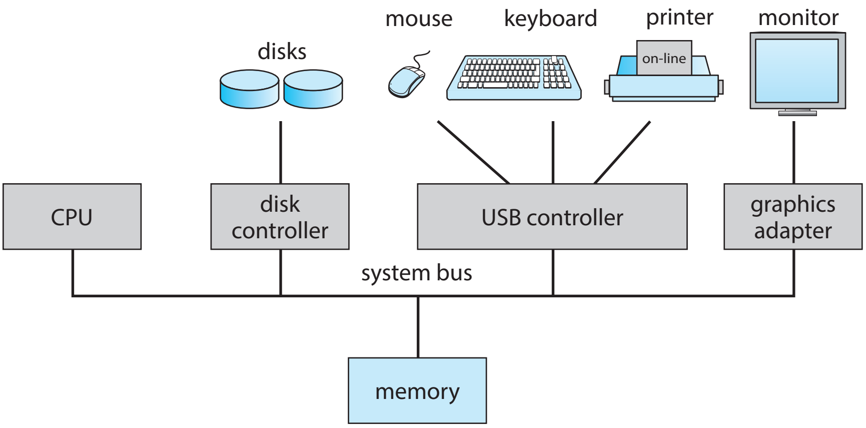


**Figure 1.2** A typical PC computer system.

### 1.2.1 Interrupts

Consider a typical computer operation: a program performing I/O. To start an I/O operation, the device driver loads the appropriate registers in the device controller. The device controller, in turn, examines the contents of these registers to determine what action to take (such as "read a character from the keyboard"). The controller starts the transfer of data from the device to its local buffer. Once the transfer of data is complete, the device controller informs the device driver that it has finished its operation. The device driver then gives control to other parts of the operating system, possibly returning the data or a pointer to the data if the operation was a read. For other operations, the device driver returns status information such as "write completed successfully" or "device busy". But how does the controller inform the device driver that it has finished its operation? This is accomplished via an **interrupt**.

#### 1.2.1.1 Overview

Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the system bus. (There may be many buses within a computer system, but the system bus is the main communications path between the major components.) Interrupts are used for many other purposes as well and are a key part of how operating systems and hardware interact.

When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location. The fixed location usually contains the starting address where the service routine for the interrupt is located. The interrupt service routine executes; on completion, the CPU resumes the interrupted computation. A timeline of this operation is shown in Figure 1.3. To run the animation assicated with this figure please click **here**.

Interrupts are an important part of a computer architecture. Each computer design has its own interrupt mechanism, but several functions are common. The interrupt must transfer control to the appropriate interrupt service routine. The straightforward method for managing this transfer would be to invoke a generic routine to examine the interrupt information. The routine, in turn,

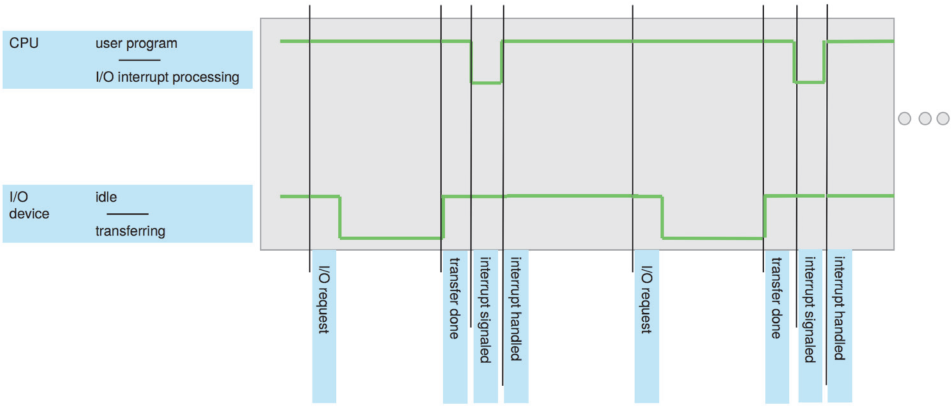


**Figure 1.3** Interrupt timeline for a single program doing output.

would call the interrupt-specific handler. However, interrupts must be handled quickly, as they occur very frequently. A table of pointers to interrupt routines can be used instead to provide the necessary speed. The interrupt routine is called indirectly through the table, with no intermediate routine needed. Generally, the table of pointers is stored in low memory (the first hundred or so locations). These locations hold the addresses of the interrupt service routines for the various devices. This array, or **interrupt vector**, of addresses is then indexed by a unique number, given with the interrupt request, to provide the address of the interrupt service routine for the interrupting device. Operating systems as different as Windows and UNIX dispatch interrupts in this manner.

The interrupt architecture must also save the state information of whatever was interrupted, so that it can restore this information after servicing the interrupt. If the interrupt routine needs to modify the processor state—for instance, by modifying register values—it must explicitly save the current state and then restore that state before returning. After the interrupt is serviced, the saved return address is loaded into the program counter, and the interrupted computation resumes as though the interrupt had not occurred.

#### 1.2.1.2 Implementation

The basic interrupt mechanism works as follows. The CPU hardware has a wire called the **interrupt-request line** that the CPU senses after executing every instruction. When the CPU detects that a controller has asserted a signal on the interrupt-request line, it reads the interrupt number and jumps to the **interrupt-handler routine** by using that interrupt number as an index into the interrupt vector. It then starts execution at the address associated with that index. The interrupt handler saves any state it will be changing during its operation, determines the cause of the interrupt, performs the necessary processing, performs a state restore, and executes a `return_from_interrupt` instruction to return the CPU to the execution state prior to the interrupt. We say that the device controller ***raises*** an interrupt by asserting a signal on the interrupt request line, the CPU ***catches*** the interrupt and ***dispatches*** it to the interrupt handler, and the handler ***clears*** the interrupt by servicing the device. Figure 1.4 summarizes the interrupt-driven I/O cycle.

The basic interrupt mechanism just described enables the CPU to respond to an asynchronous event, as when a device controller becomes ready for service. In a modern operating system, however, we need more sophisticated interrupt-handling features.

1. We need the ability to defer interrupt handling during critical processing.
2. We need an efficient way to dispatch to the proper interrupt handler for a device.
3. We need multilevel interrupts, so that the operating system can distinguish between high- and low-priority interrupts and can respond with the appropriate degree of urgency.

In modern computer hardware, these three features are provided by the CPU and the **interrupt-controller hardware**.

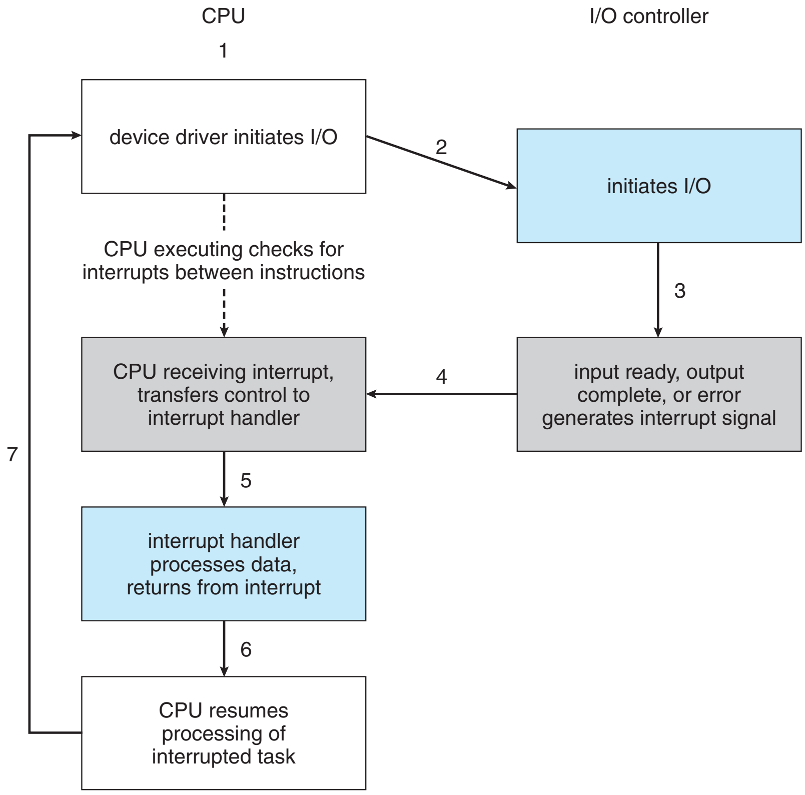

CPU
I/O controller
1
device driver initiates I/O
2
initiates I/O
CPU executing checks for interrupts between instructions
3
CPU receiving interrupt, transfers control to interrupt handler
4
input ready, output complete, or error generates interrupt signal
7
5
interrupt handler processes data, returns from interrupt
6
CPU resumes processing of interrupted task

**Figure 1.4** Interrupt-driven I/O cycle.

Most CPUs have two interrupt request lines. One is the **nonmaskable interrupt**, which is reserved for events such as unrecoverable memory errors. The second interrupt line is **maskable**: it can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted. The maskable interrupt is used by device controllers to request service.

Recall that the purpose of a vectored interrupt mechanism is to reduce the need for a single interrupt handler to search all possible sources of interrupts to determine which one needs service. In practice, however, computers have more devices (and, hence, interrupt handlers) than they have address elements in the interrupt vector. A common way to solve this problem is to use **interrupt chaining**, in which each element in the interrupt vector points to the head of a list of interrupt handlers. When an interrupt is raised, the handlers on the corresponding list are called one by one, until one is found that can service the request. This structure is a compromise between the overhead of a huge interrupt table and the inefficiency of dispatching to a single interrupt handler.

Figure 1.5 illustrates the design of the interrupt vector for Intel processors. The events from 0 to 31, which are nonmaskable, are used to signal various error conditions. The events from 32 to 255, which are maskable, are used for purposes such as device-generated interrupts.

The interrupt mechanism also implements a system of **interrupt priority levels**. These levels enable the CPU to defer the handling of low-priority inter-

| vector number | description |
|---|---|
| 0 | divide error |
| 1 | debug exception |
| 2 | null interrupt |
| 3 | breakpoint |
| 4 | INTO-detected overflow |
| 5 | bound range exception |
| 6 | invalid opcode |
| 7 | device not available |
| 8 | double fault |
| 9 | coprocessor segment overrun (reserved) |
| 10 | invalid task state segment |
| 11 | segment not present |
| 12 | stack fault |
| 13 | general protection |
| 14 | page fault |
| 15 | (Intel reserved, do not use) |
| 16 | floating-point error |
| 17 | alignment check |
| 18 | machine check |
| 19–31 | (Intel reserved, do not use) |
| 32–255 | maskable interrupts |

**Figure 1.5** Intel processor event-vector table.

rupts without masking all interrupts and makes it possible for a high-priority interrupt to preempt the execution of a low-priority interrupt.

In summary, interrupts are used throughout modern operating systems to handle asynchronous events (and for other purposes we will discuss throughout the text). Device controllers and hardware faults raise interrupts. To enable the most urgent work to be done first, modern computers use a system of interrupt priorities. Because interrupts are used so heavily for time-sensitive processing, efficient interrupt handling is required for good system performance.

### 1.2.2 Storage Structure

The CPU can load instructions only from memory, so any programs must first be loaded into memory to run. General-purpose computers run most of their programs from rewritable memory, called main memory (also called **random-access memory**, or **RAM**). Main memory commonly is implemented in a semiconductor technology called **dynamic random-access memory** (**DRAM**).

Computers use other forms of memory as well. For example, the first program to run on computer power-on is a **bootstrap program**, which then loads the operating system. Since RAM is **volatile**—loses its content when power is turned off or otherwise lost—we cannot trust it to hold the bootstrap program. Instead, for this and some other purposes, the computer uses electrically erasable programmable read-only memory (EEPROM) and other forms of **firmwar** —storage that is infrequently written to and is nonvolatile. EEPROM

### *STORAGE DEFINITIONS AND NOTATION*

The basic unit of computer storage is the **bit**. A bit can contain one of two values, 0 and 1. All other storage in a computer is based on collections of bits. Given enough bits, it is amazing how many things a computer can represent: numbers, letters, images, movies, sounds, documents, and programs, to name a few. A **byte** is 8 bits, and on most computers it is the smallest convenient chunk of storage. For example, most computers don't have an instruction to move a bit but do have one to move a byte. A less common term is **word**, which is a given computer architecture's native unit of data. A word is made up of one or more bytes. For example, a computer that has 64-bit registers and 64-bit memory addressing typically has 64-bit (8-byte) words. A computer executes many operations in its native word size rather than a byte at a time.

Computer storage, along with most computer throughput, is generally measured and manipulated in bytes and collections of bytes. A **kilobyte**, or **KB**, is 1,024 bytes; a **megabyte**, or **MB**, is $1{,}024^2$ bytes; a **gigabyte**, or **GB**, is $1{,}024^3$ bytes; a **terabyte**, or **TB**, is $1{,}024^4$ bytes; and a **petabyte**, or **PB**, is $1{,}024^5$ bytes. Computer manufacturers often round off these numbers and say that a megabyte is 1 million bytes and a gigabyte is 1 billion bytes. Networking measurements are an exception to this general rule; they are given in bits (because networks move data a bit at a time).

can be changed but cannot be changed frequently. In addition, it is low speed, and so it contains mostly static programs and data that aren't frequently used. For example, the iPhone uses EEPROM to store serial numbers and hardware information about the device.

All forms of memory provide an array of bytes. Each byte has its own address. Interaction is achieved through a sequence of `load` or `store` instructions to specific memory addresses. The `load` instruction moves a byte or word from main memory to an internal register within the CPU, whereas the `store` instruction moves the content of a register to main memory. Aside from explicit loads and stores, the CPU automatically loads instructions from main memory for execution from the location stored in the program counter.

A typical instruction–execution cycle, as executed on a system with a **von Neumann architecture**, first fetches an instruction from memory and stores that instruction in the **instruction register**. The instruction is then decoded and may cause operands to be fetched from memory and stored in some internal register. After the instruction on the operands has been executed, the result may be stored back in memory. Notice that the memory unit sees only a stream of memory addresses. It does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, or some other means) or what they are for (instructions or data). Accordingly, we can ignore ***how*** a memory address is generated by a program. We are interested only in the sequence of memory addresses generated by the running program.

Ideally, we want the programs and data to reside in main memory permanently. This arrangement usually is not possible on most systems for two reasons:

1. Main memory is usually too small to store all needed programs and data permanently.
2. Main memory, as mentioned, is volatile—it loses its contents when power is turned off or otherwise lost.

Thus, most computer systems provide **secondary storage** as an extension of main memory. The main requirement for secondary storage is that it be able to hold large quantities of data permanently.

The most common secondary-storage devices are **hard-disk drives** (**HDDs**) and **nonvolatile memory** (**NVM**) **devices**, which provide storage for both programs and data. Most programs (system and application) are stored in secondary storage until they are loaded into memory. Many programs then use secondary storage as both the source and the destination of their processing. Secondary storage is also much slower than main memory. Hence, the proper management of secondary storage is of central importance to a computer system, as we discuss in Chapter 11.

In a larger sense, however, the storage structure that we have described—consisting of registers, main memory, and secondary storage—is only one of many possible storage system designs. Other possible components include cache memory, CD-ROM or blu-ray, magnetic tapes, and so on. Those that are slow enough and large enough that they are used only for special purposes—to store backup copies of material stored on other devices, for example—are called **tertiary storage**. Each storage system provides the basic functions of storing a datum and holding that datum until it is retrieved at a later time. The main differences among the various storage systems lie in speed, size, and volatility.

The wide variety of storage systems can be organized in a hierarchy (Figure 1.6) according to storage capacity and access time. As a general rule, there is a

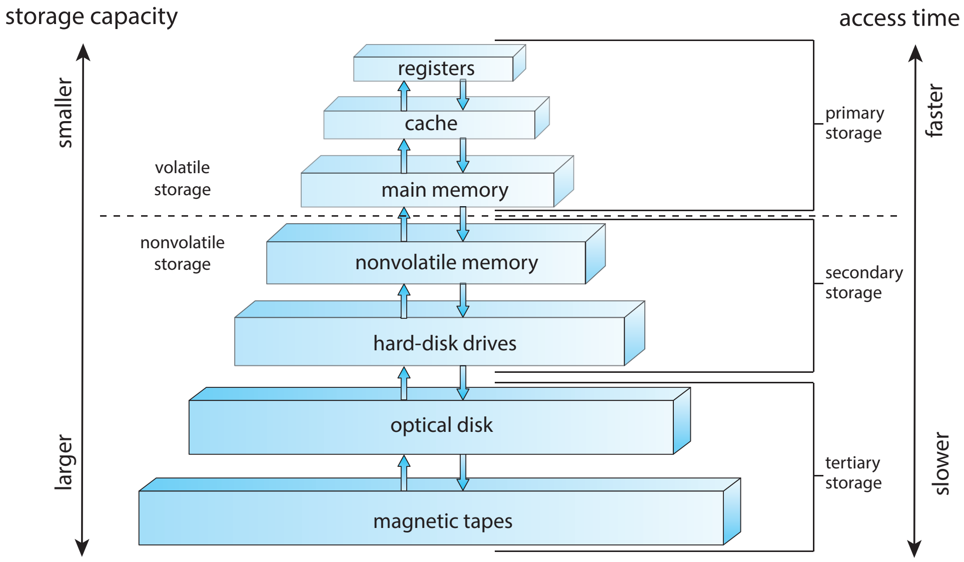

storage capacity
access time
smaller
larger
faster
slower
registers
cache
main memory
nonvolatile memory
hard-disk drives
optical disk
magnetic tapes
volatile storage
nonvolatile storage
primary storage
secondary storage
tertiary storage

**Figure 1.6** Storage-device hierarchy.

trade-off between size and speed, with smaller and faster memory closer to the CPU. As shown in the figure, in addition to differing in speed and capacity, the various storage systems are either volatile or nonvolatile. Volatile storage, as mentioned earlier, loses its contents when the power to the device is removed, so data must be written to nonvolatile storage for safekeeping.

The top four levels of memory in the figure are constructed using **semiconductor memory**, which consists of semiconductor-based electronic circuits. NVM devices, at the fourth level, have several variants but in general are faster than hard disks. The most common form of NVM device is flash memory, which is popular in mobile devices such as smartphones and tablets. Increasingly, flash memory is being used for long-term storage on laptops, desktops, and servers as well.

Since storage plays an important role in operating-system structure, we will refer to it frequently in the text. In general, we will use the following terminology:

- Volatile storage will be referred to simply as **memory**. If we need to emphasize a particular type of storage device (for example, a register),we will do so explicitly.
- Nonvolatile storage retains its contents when power is lost. It will be referred to as **NVS**. The vast majority of the time we spend on NVS will be on secondary storage. This type of storage can be classified into two distinct types:
    - **Mechanical**. A few examples of such storage systems are HDDs, optical disks, holographic storage, and magnetic tape. If we need to emphasize a particular type of mechanical storage device (for example, magnetic tape), we will do so explicitly.
    - **Electrical**. A few examples of such storage systems are flash memory, FRAM, NRAM, and SSD. Electrical storage will be referred to as **NVM**. If we need to emphasize a particular type of electrical storage device (for example, SSD), we will do so explicitly.

  Mechanical storage is generally larger and less expensive per byte than electrical storage. Conversely, electrical storage is typically costly, smaller, and faster than mechanical storage.

The design of a complete storage system must balance all the factors just discussed: it must use only as much expensive memory as necessary while providing as much inexpensive, nonvolatile storage as possible. Caches can be installed to improve performance where a large disparity in access time or transfer rate exists between two components.

### 1.2.3 I/O Structure

A large portion of operating system code is dedicated to managing I/O, both because of its importance to the reliability and performance of a system and because of the varying nature of the devices.

Recall from the beginning of this section that a general-purpose computer system consists of multiple devices, all of which exchange data via a common

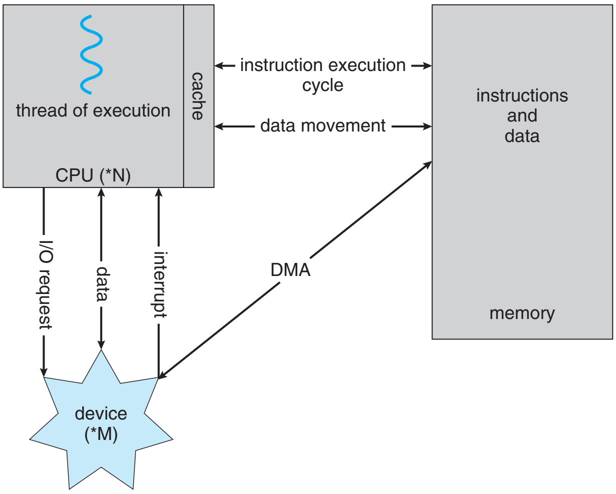


**Figure 1.7** How a modern computer system works.

bus. The form of interrupt-driven I/O described in Section 1.2.1 is fine for moving small amounts of data but can produce high overhead when used for bulk data movement such as NVS I/O. To solve this problem, **direct memory access** (**DMA**) is used. After setting up buffers, pointers, and counters for the I/O device, the device controller transfers an entire block of data directly to or from the device and main memory, with no intervention by the CPU. Only one interrupt is generated per block, to tell the device driver that the operation has completed, rather than the one interrupt per byte generated for low-speed devices. While the device controller is performing these operations, the CPU is available to accomplish other work.

Some high-end systems use switch rather than bus architecture. On these systems, multiple components can talk to other components concurrently, rather than competing for cycles on a shared bus. In this case, DMA is even more effective. Figure 1.7 shows the interplay of all components of a computer system.

## 1.3 Computer-System Architecture

In Section 1.2, we introduced the general structure of a typical computer system. A computer system can be organized in a number of different ways, which we can categorize roughly according to the number of general-purpose processors used.

### 1.3.1 Single-Processor Systems

Many years ago, most computer systems used a single processor containing one CPU with a single processing core. The **core** is the component that executes instructions and registers for storing data locally. The one main CPU with its core is capable of executing a general-purpose instruction set, including instructions from processes. These systems have other special-purpose processors as well. They may come in the form of device-specific processors, such as disk, keyboard, and graphics controllers.

All of these special-purpose processors run a limited instruction set and do not run processes. Sometimes, they are managed by the operating system, in that the operating system sends them information about their next task and monitors their status. For example, a disk-controller microprocessor receives a sequence of requests from the main CPU core and implements its own disk queue and scheduling algorithm. This arrangement relieves the main CPU of the overhead of disk scheduling. PCs contain a microprocessor in the keyboard to convert the keystrokes into codes to be sent to the CPU. In other systems or circumstances, special-purpose processors are low-level components built into the hardware. The operating system cannot communicate with these processors; they do their jobs autonomously. The use of special-purpose microprocessors is common and does not turn a single-processor system into a multiprocessor. If there is only one general-purpose CPU with a single processing core, then the system is a single-processor system. According to this definition, however, very few contemporary computer systems are single-processor systems.

### 1.3.2 Multiprocessor Systems

On modern computers, from mobile devices to servers, **multiprocessor systems** now dominate the landscape of computing. Traditionally, such systems have two (or more) processors, each with a single-core CPU. The processors share the computer bus and sometimes the clock, memory, and peripheral devices. The primary advantage of multiprocessor systems is increased throughput. That is, by increasing the number of processors, we expect to get more work done in less time. The speed-up ratio with $N$ processors is not $N$, however; it is less than $N$. When multiple processors cooperate on a task, a certain amount of overhead is incurred in keeping all the parts working correctly. This overhead, plus contention for shared resources, lowers the expected gain from additional processors.

The most common multiprocessor systems use **symmetric multiprocessing** (**SMP**), in which each peer CPU processor performs all tasks, including operating-system functions and user processes. Figure 1.8 illustrates a typical SMP architecture with two processors, each with its own CPU. Notice that each CPU processor has its own set of registers, as well as a private—or local—cache. However, all processors share physical memory over the system bus.

The benefit of this model is that many processes can run simultaneously —$N$ processes can run if there are $N$ CPUs—without causing performance to deteriorate significantly. However, since the CPUs are separate, one may be sitting idle while another is overloaded, resulting in inefficiencies. These inefficiencies can be avoided if the processors share certain data structures. A multiprocessor system of this form will allow processes and resources—such as memory—to be shared dynamically among the various processors and can lower the workload variance among the processors. Such a system must be written carefully, as we shall see in Chapter 5 and Chapter 6.

The definition of ***multiprocessor*** has evolved over time and now includes **multicore** systems, in which multiple computing cores reside on a single chip. Multicore systems can be more efficient than multiple chips with single cores because on-chip communication is faster than between-chip communication.

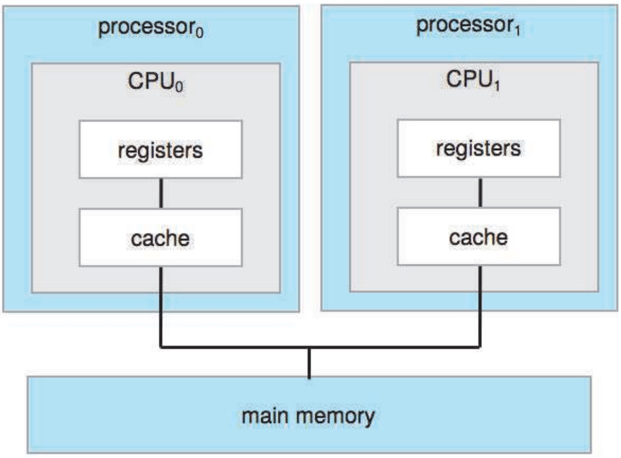

processor₀
CPU₀
registers
cache
processor₁
CPU₁
registers
cache
main memory

**Figure 1.8** Symmetric multiprocessing architecture.

In addition, one chip with multiple cores uses significantly less power than multiple single-core chips, an important issue for mobile devices as well as laptops.

In Figure 1.9, we show a dual-core design with two cores on the same processor chip. In this design, each core has its own register set, as well as its own local cache, often known as a level 1, or L1, cache. Notice, too, that a level 2 (L2) cache is local to the chip but is shared by the two processing cores. Most architectures adopt this approach, combining local and shared caches, where local, lower-level caches are generally smaller and faster than higher-level shared

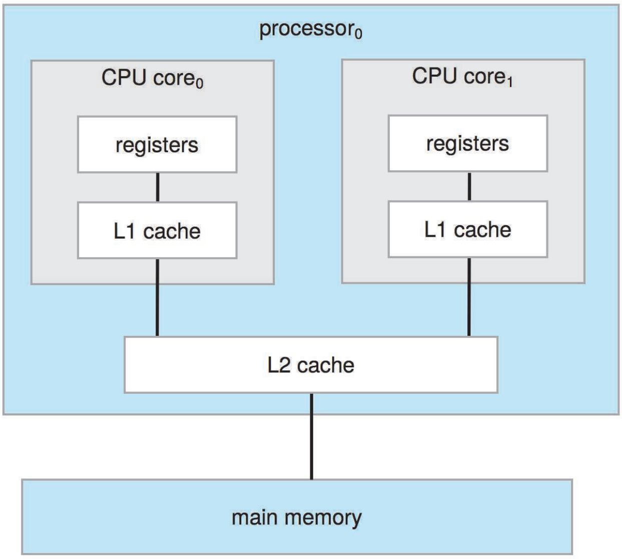

processor₀
CPU core₀
registers
L1 cache
CPU core₁
registers
L1 cache
L2 cache
main memory

**Figure 1.9** A dual-core design with two cores on the same chip.

### *DEFINITIONS OF COMPUTER SYSTEM COMPONENTS*

- **CPU**—The hardware that executes instructions.
- **Processor**—A physical chip that contains one or more CPUs.
- **Core**—The basic computation unit of the CPU.
- **Multicore**—Including multiple computing cores on the same CPU.
- **Multiprocessor**—Including multiple processors.

Although virtually all systems are now multicore, we use the general term ***CPU*** when referring to a single computational unit of a computer system and ***core*** as well as ***multicore*** when specifically referring to one or more cores on a CPU.

caches. Aside from architectural considerations, such as cache, memory, and bus contention, a multicore processor with *N* cores appears to the operating system as *N* standard CPUs. This characteristic puts pressure on operating-system designers—and application programmers—to make efficient use of these processing cores, an issue we pursue in Chapter 4. Virtually all modern operating systems—including Windows, macOS, and Linux, as well as Android and iOS mobile systems—support multicore SMP systems.

Adding additional CPUs to a multiprocessor system will increase computing power; however, as suggested earlier, the concept does not scale very well, and once we add too many CPUs, contention for the system bus becomes a bottleneck and performance begins to degrade. An alternative approach is instead to provide each CPU (or group of CPUs) with its own local memory that is accessed via a small, fast local bus. The CPUs are connected by a **shared system interconnect**, so that all CPUs share one physical address space. This approach—known as **non-uniform memory access**, or **NUMA**—is illustrated in Figure 1.10. The advantage is that, when a CPU accesses its local memory, not only is it fast, but there is also no contention over the system interconnect. Thus, NUMA systems can scale more effectively as more processors are added.

A potential drawback with a NUMA system is increased latency when a CPU must access remote memory across the system interconnect, creating a possible performance penalty. In other words, for example, $\text{CPU}_0$ cannot access the local memory of $\text{CPU}_3$ as quickly as it can access its own local memory, slowing down performance. Operating systems can minimize this NUMA penalty through careful CPU scheduling and memory management, as discussed in Section 5.5.2 and Section 10.5.4. Because NUMA systems can scale to accommodate a large number of processors, they are becoming increasingly popular on servers as well as high-performance computing systems.

Finally, **blade servers** are systems in which multiple processor boards, I/O boards, and networking boards are placed in the same chassis. The difference between these and traditional multiprocessor systems is that each blade-processor board boots independently and runs its own operating system. Some blade-server boards are multiprocessor as well, which blurs the lines between

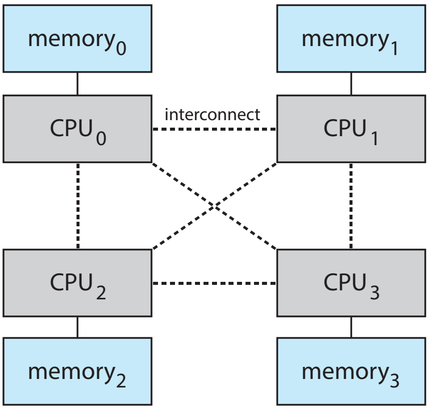


**Figure 1.10** NUMA multiprocessing architecture.

types of computers. In essence, these servers consist of multiple independent multiprocessor systems.

### 1.3.3 Clustered Systems

Another type of multiprocessor system is a **clustered system**, which gathers together multiple CPUs. Clustered systems differ from the multiprocessor systems described in Section 1.3.2 in that they are composed of two or more individual systems—or nodes—joined together; each node is typically a multicore system. Such systems are considered **loosely coupled**. We should note that the definition of ***clustered*** is not concrete; many commercial and open-source packages wrestle to define what a clustered system is and why one form is better than another. The generally accepted definition is that clustered computers share storage and are closely linked via a local-area network LAN (as described in Chapter 19) or a faster interconnect, such as InfiniBand.

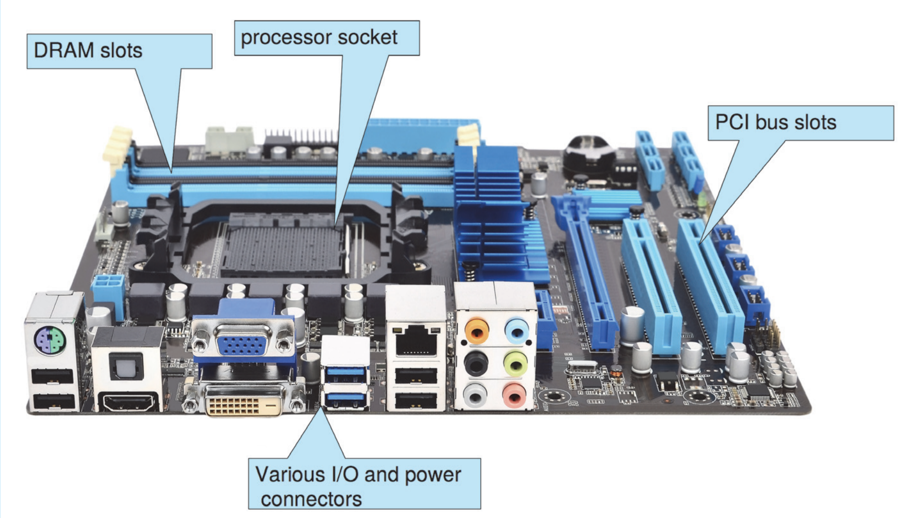
Clustering is usually used to provide **high-availability service**—that is, service that will continue even if one or more systems in the cluster fail. Generally, we obtain high availability by adding a level of redundancy in the system. A layer of cluster software runs on the cluster nodes. Each node can monitor one or more of the others (over the network). If the monitored machine fails, the monitoring machine can take ownership of its storage and restart the applications that were running on the failed machine. The users and clients of the applications see only a brief interruption of service.

High availability provides increased reliability, which is crucial in many applications. The ability to continue providing service proportional to the level of surviving hardware is called **graceful degradation**. Some systems go beyond graceful degradation and are called **fault tolerant**, because they can suffer a failure of any single component and still continue operation. Fault tolerance requires a mechanism to allow the failure to be detected, diagnosed, and, if possible, corrected.

Clustering can be structured asymmetrically or symmetrically. In **asymmetric clustering**, one machine is in **hot-standby mode** while the other is running the applications. The hot-standby host machine does nothing but monitor the active server. If that server fails, the hot-standby host becomes the active

### *PC MOTHERBOARD*

Consider the desktop PC motherboard with a processor socket shown below:



This board is a fully functioning computer, once its slots are populated. It consists of a processor socket containing a CPU, DRAM sockets, PCIe bus slots, and I/O connectors of various types. Even the lowest-cost general-purpose CPU contains multiple cores. Some motherboards contain multiple processor sockets. More advanced computers allow more than one system board, creating NUMA systems.

server. In **symmetric clustering**, two or more hosts are running applications and are monitoring each other. This structure is obviously more efficient, as it uses all of the available hardware. However, it does require that more than one application be available to run.

Since a cluster consists of several computer systems connected via a network, clusters can also be used to provide **high-performance computing** environments. Such systems can supply significantly greater computational power than single-processor or even SMP systems because they can run an application concurrently on all computers in the cluster. The application must have been written specifically to take advantage of the cluster, however. This involves a technique known as **parallelization**, which divides a program into separate components that run in parallel on individual cores in a computer or computers in a cluster. Typically, these applications are designed so that once each computing node in the cluster has solved its portion of the problem, the results from all the nodes are combined into a final solution.

Other forms of clusters include parallel clusters and clustering over a wide-area network (WAN) (as described in Chapter 19). Parallel clusters allow multiple hosts to access the same data on shared storage. Because most oper-

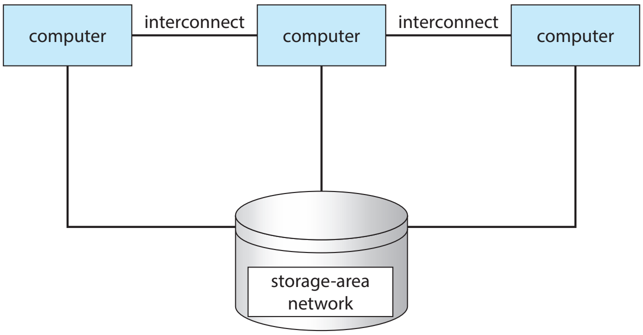


**Figure 1.11** General structure of a clustered system.

ating systems lack support for simultaneous data access by multiple hosts, parallel clusters usually require the use of special versions of software and special releases of applications. For example, Oracle Real Application Cluster is a version of Oracle's database that has been designed to run on a parallel cluster. Each machine runs Oracle, and a layer of software tracks access to the shared disk. Each machine has full access to all data in the database. To provide this shared access, the system must also supply access control and locking to ensure that no conflicting operations occur. This function, commonly known as a **distributed lock manager** (**DLM**), is included in some cluster technology.

Cluster technology is changing rapidly. Some cluster products support thousands of systems in a cluster, as well as clustered nodes that are separated by miles. Many of these improvements are made possible by **storage-area networks** (**SANs**), as described in Section 11.7.4, which allow many systems to attach to a pool of storage. If the applications and their data are stored on the SAN, then the cluster software can assign the application to run on any host that is attached to the SAN. If the host fails, then any other host can take over. In a database cluster, dozens of hosts can share the same database, greatly increasing performance and reliability. Figure 1.11 depicts the general structure of a clustered system.

## 1.4 Operating-System Operations

Now that we have discussed basic information about computer-system organization and architecture, we are ready to talk about operating systems. An operating system provides the environment within which programs are executed. Internally, operating systems vary greatly, since they are organized along many different lines. There are, however, many commonalities, which we consider in this section.

For a computer to start running—for instance, when it is powered up or rebooted—it needs to have an initial program to run. As noted earlier, this initial program, or bootstrap program, tends to be simple. Typically, it is stored within the computer hardware in firmware. It initializes all aspects of the system, from CPU registers to device controllers to memory contents. The bootstrap program must know how to load the operating system and how to

### *HADOOP*

Hadoop is an open-source software framework that is used for distributed processing of large data sets (known as **big data**) in a clustered system containing simple, low-cost hardware components. Hadoop is designed to scale from a single system to a cluster containing thousands of computing nodes. Tasks are assigned to a node in the cluster, and Hadoop arranges communication between nodes to manage parallel computations to process and coalesce results. Hadoop also detects and manages failures in nodes, providing an efficient and highly reliable distributed computing service.

Hadoop is organized around the following three components:

1. A distributed file system that manages data and files across distributed computing nodes.
2. The YARN ("Yet Another Resource Negotiator") framework, which manages resources within the cluster as well as scheduling tasks on nodes in the cluster.
3. The **MapReduce** system, which allows parallel processing of data across nodes in the cluster.

Hadoop is designed to run on Linux systems, and Hadoop applications can be written using several programming languages, including scripting languages such as PHP, Perl, and Python. Java is a popular choice for developing Hadoop applications, as Hadoop has several Java libraries that support MapReduce. More information on MapReduce and Hadoop can be found at https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html and https://hadoop.apache.org

start executing that system. To accomplish this goal, the bootstrap program must locate the operating-system kernel and load it into memory.

Once the kernel is loaded and executing, it can start providing services to the system and its users. Some services are provided outside of the kernel by system programs that are loaded into memory at boot time to become **system daemons**, which run the entire time the kernel is running. On Linux, the first system program is "`systemd`," and it starts many other daemons. Once this phase is complete, the system is fully booted, and the system waits for some event to occur.

If there are no processes to execute, no I/O devices to service, and no users to whom to respond, an operating system will sit quietly, waiting for something to happen. Events are almost always signaled by the occurrence of an interrupt. In Section 1.2.1 we described hardware interrupts. Another form of interrupt is a **trap** (or an **exception**), which is a software-generated interrupt caused either by an error (for example, division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed by executing a special operation called a **system call**.

### 1.4.1 Multiprogramming and Multitasking

One of the most important aspects of operating systems is the ability to run multiple programs, as a single program cannot, in general, keep either the CPU or the I/O devices busy at all times. Furthermore, users typically *want* to run more than one program at a time as well. **Multiprogramming** increases CPU utilization, as well as keeping users satisfied, by organizing programs so that the CPU always has one to execute. In a multiprogrammed system, a program in execution is termed a **process**.

The idea is as follows: The operating system keeps several processes in memory simultaneously (Figure 1.12). The operating system picks and begins to execute one of these processes. Eventually, the process may have to wait for some task, such as an I/O operation, to complete. In a non-multiprogrammed system, the CPU would sit idle. In a multiprogrammed system, the operating system simply switches to, and executes, another process. When ***that*** process needs to wait, the CPU switches to ***another*** process, and so on. Eventually, the first process finishes waiting and gets the CPU back. As long as at least one process needs to execute, the CPU is never idle.

This idea is common in other life situations. A lawyer does not work for only one client at a time, for example. While one case is waiting to go to trial or have papers typed, the lawyer can work on another case. If she has enough clients, the lawyer will never be idle for lack of work. (Idle lawyers tend to become politicians, so there is a certain social value in keeping lawyers busy.)

**Multitasking** is a logical extension of multiprogramming. In multitasking systems, the CPU executes multiple processes by switching among them, but the switches occur frequently, providing the user with a fast **response time**. Consider that when a process executes, it typically executes for only a short time before it either finishes or needs to perform I/O. I/O may be interactive; that is, output goes to a display for the user, and input comes from a user keyboard, mouse, or touch screen. Since interactive I/O typically runs at “people speeds,” it may take a long time to complete. Input, for example, may be

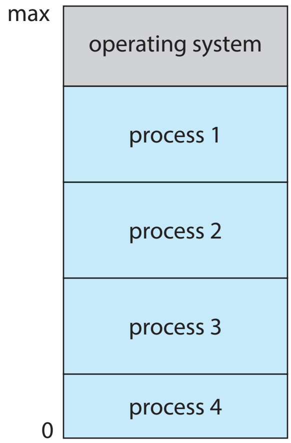


**Figure 1.12** Memory layout for a multiprogramming system.

bounded by the user's typing speed; seven characters per second is fast for people but incredibly slow for computers. Rather than let the CPU sit idle as this interactive input takes place, the operating system will rapidly switch the CPU to another process.

Having several processes in memory at the same time requires some form of memory management, which we cover in Chapter 9 and Chapter 10. In addition, if several processes are ready to run at the same time, the system must choose which process will run next. Making this decision is **CPU scheduling**, which is discussed in Chapter 5. Finally, running multiple processes concurrently requires that their ability to affect one another be limited in all phases of the operating system, including process scheduling, disk storage, and memory management. We discuss these considerations throughout the text.

In a multitasking system, the operating system must ensure reasonable response time. A common method for doing so is **virtual memory**, a technique that allows the execution of a process that is not completely in memory (Chapter 10). The main advantage of this scheme is that it enables users to run programs that are larger than actual **physical memory**. Further, it abstracts main memory into a large, uniform array of storage, separating **logical memory** as viewed by the user from physical memory. This arrangement frees programmers from concern over memory-storage limitations.

Multiprogramming and multitasking systems must also provide a file system (Chapter 13, Chapter 14, and Chapter 15). The file system resides on a secondary storage; hence, storage management must be provided (Chapter 11). In addition, a system must protect resources from inappropriate use (Chapter 17). To ensure orderly execution, the system must also provide mechanisms for process synchronization and communication (Chapter 6 and Chapter 7), and it may ensure that processes do not get stuck in a deadlock, forever waiting for one another (Chapter 8).

### 1.4.2 Dual-Mode and Multimode Operation

Since the operating system and its users share the hardware and software resources of the computer system, a properly designed operating system must ensure that an incorrect (or malicious) program cannot cause other programs—or the operating system itself—to execute incorrectly. In order to ensure the proper execution of the system, we must be able to distinguish between the execution of operating-system code and user-defined code. The approach taken by most computer systems is to provide hardware support that allows differentiation among various modes of execution.

At the very least, we need two separate ***modes*** of operation: **user mode** and **kernel mode** (also called **supervisor mode**, **system mode**, or **privileged mode**). A bit, called the **mode bit**, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1). With the mode bit, we can distinguish between a task that is executed on behalf of the operating system and one that is executed on behalf of the user. When the computer system is executing on behalf of a user application, the system is in user mode. However, when a user application requests a service from the operating system (via a system call), the system must transition from user to kernel mode to fulfill

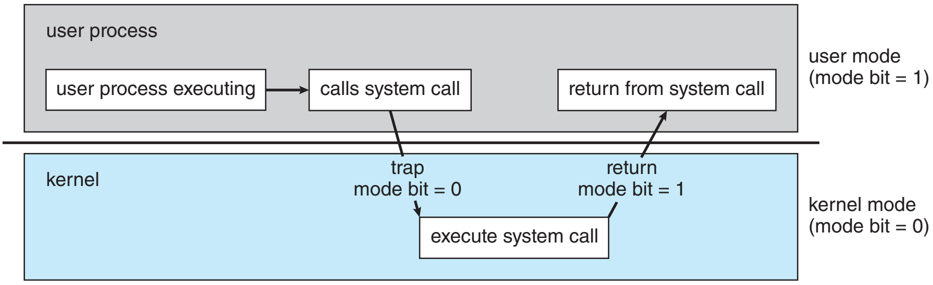


**Figure 1.13** Transition from user to kernel mode.

the request. This is shown in Figure 1.13. As we shall see, this architectural enhancement is useful for many other aspects of system operation as well.

At system boot time, the hardware starts in kernel mode. The operating system is then loaded and starts user applications in user mode. Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode (that is, changes the state of the mode bit to 0). Thus, whenever the operating system gains control of the computer, it is in kernel mode. The system always switches to user mode (by setting the mode bit to 1) before passing control to a user program.

The dual mode of operation provides us with the means for protecting the operating system from errant users—and errant users from one another. We accomplish this protection by designating some of the machine instructions that may cause harm as **privileged instructions**. The hardware allows privileged instructions to be executed only in kernel mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the operating system.

The instruction to switch to kernel mode is an example of a privileged instruction. Some other examples include I/O control, timer management, and interrupt management. Many additional privileged instructions are discussed throughout the text.

The concept of modes can be extended beyond two modes. For example, Intel processors have four separate **protection rings**, where ring 0 is kernel mode and ring 3 is user mode. (Although rings 1 and 2 could be used for various operating-system services, in practice they are rarely used.) ARMv8 systems have seven modes. CPUs that support virtualization (Section 18.1) frequently have a separate mode to indicate when the **virtual machine manager** (**VMM**) is in control of the system. In this mode, the VMM has more privileges than user processes but fewer than the kernel. It needs that level of privilege so it can create and manage virtual machines, changing the CPU state to do so.

We can now better understand the life cycle of instruction execution in a computer system. Initial control resides in the operating system, where instructions are executed in kernel mode. When control is given to a user application, the mode is set to user mode. Eventually, control is switched back to the operating system via an interrupt, a trap, or a system call. Most contemporary operating systems—such as Microsoft Windows, Unix, and Linux—

take advantage of this dual-mode feature and provide greater protection for the operating system.

System calls provide the means for a user program to ask the operating system to perform tasks reserved for the operating system on the user program's behalf. A system call is invoked in a variety of ways, depending on the functionality provided by the underlying processor. In all forms, it is the method used by a process to request action by the operating system. A system call usually takes the form of a trap to a specific location in the interrupt vector. This trap can be executed by a generic `trap` instruction, although some systems have a specific `syscall` instruction to invoke a system call.

When a system call is executed, it is typically treated by the hardware as a software interrupt. Control passes through the interrupt vector to a service routine in the operating system, and the mode bit is set to kernel mode. The system-call service routine is a part of the operating system. The kernel examines the interrupting instruction to determine what system call has occurred; a parameter indicates what type of service the user program is requesting. Additional information needed for the request may be passed in registers, on the stack, or in memory (with pointers to the memory locations passed in registers). The kernel verifies that the parameters are correct and legal, executes the request, and returns control to the instruction following the system call. We describe system calls more fully in Section 2.3.

Once hardware protection is in place, it detects errors that violate modes. These errors are normally handled by the operating system. If a user program fails in some way—such as by making an attempt either to execute an illegal instruction or to access memory that is not in the user's address space—then the hardware traps to the operating system. The trap transfers control through the interrupt vector to the operating system, just as an interrupt does. When a program error occurs, the operating system must terminate the program abnormally. This situation is handled by the same code as a user-requested abnormal termination. An appropriate error message is given, and the memory of the program may be dumped. The memory dump is usually written to a file so that the user or programmer can examine it and perhaps correct it and restart the program.

### 1.4.3 Timer

We must ensure that the operating system maintains control over the CPU. We cannot allow a user program to get stuck in an infinite loop or to fail to call system services and never return control to the operating system. To accomplish this goal, we can use a **timer**. A timer can be set to interrupt the computer after a specified period. The period may be fixed (for example, 1/60 second) or variable (for example, from 1 millisecond to 1 second). A **variable timer** is generally implemented by a fixed-rate clock and a counter. The operating system sets the counter. Every time the clock ticks, the counter is decremented. When the counter reaches 0, an interrupt occurs. For instance, a 10-bit counter with a 1-millisecond clock allows interrupts at intervals from 1 millisecond to 1,024 milliseconds, in steps of 1 millisecond.

Before turning over control to the user, the operating system ensures that the timer is set to interrupt. If the timer interrupts, control transfers automatically to the operating system, which may treat the interrupt as a fatal error or

### *LINUX TIMERS*

On Linux systems, the kernel configuration parameter `HZ` specifies the frequency of timer interrupts. An `HZ` value of 250 means that the timer generates 250 interrupts per second, or one interrupt every 4 milliseconds. The value of `HZ` depends upon how the kernel is configured, as well the machine type and architecture on which it is running. A related kernel variable is `jiffies`, which represent the number of timer interrupts that have occurred since the system was booted. A programming project in Chapter 2 further explores timing in the Linux kernel.

may give the program more time. Clearly, instructions that modify the content of the timer are privileged.

## 1.5 Resource Management

As we have seen, an operating system is a **resource manager**. The system's CPU, memory space, file-storage space, and I/O devices are among the resources that the operating system must manage.

### 1.5.1 Process Management

A program can do nothing unless its instructions are executed by a CPU. A program in execution, as mentioned, is a process. A program such as a compiler is a process, and a word-processing program being run by an individual user on a PC is a process. Similarly, a social media app on a mobile device is a process. For now, you can consider a process to be an instance of a program in execution, but later you will see that the concept is more general. As described in Chapter 3, it is possible to provide system calls that allow processes to create subprocesses to execute concurrently.

A process needs certain resources—including CPU time, memory, files, and I/O devices—to accomplish its task. These resources are typically allocated to the process while it is running. In addition to the various physical and logical resources that a process obtains when it is created, various initialization data (input) may be passed along. For example, consider a process running a web browser whose function is to display the contents of a web page on a screen. The process will be given the URL as an input and will execute the appropriate instructions and system calls to obtain and display the desired information on the screen. When the process terminates, the operating system will reclaim any reusable resources.

We emphasize that a program by itself is not a process. A program is a ***passive*** entity, like the contents of a file stored on disk, whereas a process is an ***active*** entity. A single-threaded process has one **program counter** specifying the next instruction to execute. (Threads are covered in Chapter 4.) The execution of such a process must be sequential. The CPU executes one instruction of the process after another, until the process completes. Further, at any time, one instruction at most is executed on behalf of the process. Thus, although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. A multithreaded process has multiple program counters, each pointing to the next instruction to execute for a given thread.

A process is the unit of work in a system. A system consists of a collection of processes, some of which are operating-system processes (those that execute system code) and the rest of which are user processes (those that execute user code). All these processes can potentially execute concurrently—by multiplexing on a single CPU core—or in parallel across multiple CPU cores.

The operating system is responsible for the following activities in connection with process management:

- Creating and deleting both user and system processes
- Scheduling processes and threads on the CPUs
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication

We discuss process-management techniques in Chapter 3 through Chapter 7.

### 1.5.2 Memory Management

As discussed in Section 1.2.2, the main memory is central to the operation of a modern computer system. Main memory is a large array of bytes, ranging in size from hundreds of thousands to billions. Each byte has its own address. Main memory is a repository of quickly accessible data shared by the CPU and I/O devices. The CPU reads instructions from main memory during the instruction-fetch cycle and both reads and writes data from main memory during the data-fetch cycle (on a von Neumann architecture). As noted earlier, the main memory is generally the only large storage device that the CPU is able to address and access directly. For example, for the CPU to process data from disk, those data must first be transferred to main memory by CPU-generated I/O calls. In the same way, instructions must be in memory for the CPU to execute them.

For a program to be executed, it must be mapped to absolute addresses and loaded into memory. As the program executes, it accesses program instructions and data from memory by generating these absolute addresses. Eventually, the program terminates, its memory space is declared available, and the next program can be loaded and executed.

To improve both the utilization of the CPU and the speed of the computer's response to its users, general-purpose computers must keep several programs in memory, creating a need for memory management. Many different memory-management schemes are used. These schemes reflect various approaches, and the effectiveness of any given algorithm depends on the situation. In selecting a memory-management scheme for a specific system, we must take into account many factors—especially the ***hardware*** design of the system. Each algorithm requires its own hardware support.

The operating system is responsible for the following activities in connection with memory management:

- Keeping track of which parts of memory are currently being used and which process is using them
- Allocating and deallocating memory space as needed
- Deciding which processes (or parts of processes) and data to move into and out of memory

Memory-management techniques are discussed in Chapter 9 and Chapter 10.

### 1.5.3 File-System Management

To make the computer system convenient for users, the operating system provides a uniform, logical view of information storage. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the **file**. The operating system maps files onto physical media and accesses these files via the storage devices.

File management is one of the most visible components of an operating system. Computers can store information on several different types of physical media. Secondary storage is the most common, but tertiary storage is also possible. Each of these media has its own characteristics and physical organization. Most are controlled by a device, such as a disk drive, that also has its own unique characteristics. These properties include access speed, capacity, data-transfer rate, and access method (sequential or random).

A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be free-form (for example, text files), or they may be formatted rigidly (for example, fixed fields such as an mp3 music file). Clearly, the concept of a file is an extremely general one.

The operating system implements the abstract concept of a file by managing mass storage media and the devices that control them. In addition, files are normally organized into directories to make them easier to use. Finally, when multiple users have access to files, it may be desirable to control which user may access a file and how that user may access it (for example, read, write, append).

The operating system is responsible for the following activities in connection with file management:

- Creating and deleting files
- Creating and deleting directories to organize files
- Supporting primitives for manipulating files and directories
- Mapping files onto mass storage
- Backing up files on stable (nonvolatile) storage media

File-management techniques are discussed in Chapter 13, Chapter 14, and Chapter 15.

### 1.5.4 Mass-Storage Management

As we have already seen, the computer system must provide secondary storage to back up main memory. Most modern computer systems use HDDs and NVM devices as the principal on-line storage media for both programs and data. Most programs—including compilers, web browsers, word processors, and games—are stored on these devices until loaded into memory. The programs then use the devices as both the source and the destination of their processing. Hence, the proper management of secondary storage is of central importance to a computer system. The operating system is responsible for the following activities in connection with secondary storage management:

- Mounting and unmounting
- Free-space management
- Storage allocation
- Disk scheduling
- Partitioning
- Protection

Because secondary storage is used frequently and extensively, it must be used efficiently. The entire speed of operation of a computer may hinge on the speeds of the secondary storage subsystem and the algorithms that manipulate that subsystem.

At the same time, there are many uses for storage that is slower and lower in cost (and sometimes higher in capacity) than secondary storage. Backups of disk data, storage of seldom-used data, and long-term archival storage are some examples. Magnetic tape drives and their tapes and CD DVD and Blu-ray drives and platters are typical tertiary storage devices.

Tertiary storage is not crucial to system performance, but it still must be managed. Some operating systems take on this task, while others leave tertiary-storage management to application programs. Some of the functions that operating systems can provide include mounting and unmounting media in devices, allocating and freeing the devices for exclusive use by processes, and migrating data from secondary to tertiary storage.

Techniques for secondary storage and tertiary storage management are discussed in Chapter 11.

### 1.5.5 Cache Management

**Caching** is an important principle of computer systems. Here's how it works. Information is normally kept in some storage system (such as main memory). As it is used, it is copied into a faster storage system—the cache—on a temporary basis. When we need a particular piece of information, we first check whether it is in the cache. If it is, we use the information directly from the cache.

| Level | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Name | registers | cache | main memory | solid-state disk | magnetic disk |
| Typical size | < 1 KB | < 16MB | < 64GB | < 1 TB | < 10 TB |
| Implementation technology | custom memory with multiple ports CMOS | on-chip or off-chip CMOS SRAM | CMOS SRAM | flash memory | magnetic disk |
| Access time (ns) | 0.25-0.5 | 0.5-25 | 80-250 | 25,000-50,000 | 5,000,000 |
| Bandwidth (MB/sec) | 20,000-100,000 | 5,000-10,000 | 1,000-5,000 | 500 | 20-150 |
| Managed by | compiler | hardware | operating system | operating system | operating system |
| Backed by | cache | main memory | disk | disk | disk or tape |

**Figure 1.14** Characteristics of various types of storage.

If it is not, we use the information from the source, putting a copy in the cache under the assumption that we will need it again soon.

In addition, internal programmable registers provide a high-speed cache for main memory. The programmer (or compiler) implements the register-allocation and register-replacement algorithms to decide which information to keep in registers and which to keep in main memory.

Other caches are implemented totally in hardware. For instance, most systems have an instruction cache to hold the instructions expected to be executed next. Without this cache, the CPU would have to wait several cycles while an instruction was fetched from main memory. For similar reasons, most systems have one or more high-speed data caches in the memory hierarchy. We are not concerned with these hardware-only caches in this text, since they are outside the control of the operating system.

Because caches have limited size, **cache management** is an important design problem. Careful selection of the cache size and of a replacement policy can result in greatly increased performance, as you can see by examining Figure 1.14. Replacement algorithms for software-controlled caches are discussed in Chapter 10.

The movement of information between levels of a storage hierarchy may be either explicit or implicit, depending on the hardware design and the controlling operating-system software. For instance, data transfer from cache to CPU and registers is usually a hardware function, with no operating-system intervention. In contrast, transfer of data from disk to memory is usually controlled by the operating system.

In a hierarchical storage structure, the same data may appear in different levels of the storage system. For example, suppose that an integer A that is to be incremented by 1 is located in file B, and file B resides on hard disk. The increment operation proceeds by first issuing an I/O operation to copy the disk block on which A resides to main memory. This operation is followed by copying A to the cache and to an internal register. Thus, the copy of A appears in several places: on the hard disk, in main memory, in the cache, and in an internal register (see Figure 1.15). Once the increment takes place in the internal register, the value of A differs in the various storage systems. The value of A

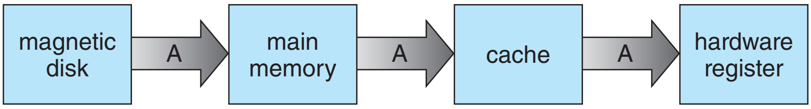


**Figure 1.15** Migration of integer A from disk to register.

becomes the same only after the new value of A is written from the internal register back to the hard disk.

In a computing environment where only one process executes at a time, this arrangement poses no difficulties, since an access to integer A will always be to the copy at the highest level of the hierarchy. However, in a multitasking environment, where the CPU is switched back and forth among various processes, extreme care must be taken to ensure that, if several processes wish to access A, then each of these processes will obtain the most recently updated value of A.

The situation becomes more complicated in a multiprocessor environment where, in addition to maintaining internal registers, each of the CPUs also contains a local cache (refer back to Figure 1.8). In such an environment, a copy of A may exist simultaneously in several caches. Since the various CPUs can all execute in parallel, we must make sure that an update to the value of A in one cache is immediately reflected in all other caches where A resides. This situation is called **cache coherency**, and it is usually a hardware issue (handled below the operating-system level).

In a distributed environment, the situation becomes even more complex. In this environment, several copies (or replicas) of the same file can be kept on different computers. Since the various replicas may be accessed and updated concurrently, some distributed systems ensure that, when a replica is updated in one place, all other replicas are brought up to date as soon as possible. There are various ways to achieve this guarantee, as we discuss in Chapter 19.

### 1.5.6 I/O System Management

One of the purposes of an operating system is to hide the peculiarities of specific hardware devices from the user. For example, in UNIX, the peculiarities of I/O devices are hidden from the bulk of the operating system itself by the **I/O subsystem**. The I/O subsystem consists of several components:

- A memory-management component that includes buffering, caching, and spooling
- A general device-driver interface
- Drivers for specific hardware devices

Only the device driver knows the peculiarities of the specific device to which it is assigned.

We discussed earlier in this chapter how interrupt handlers and device drivers are used in the construction of efficient I/O subsystems. In Chapter 12, we discuss how the I/O subsystem interfaces to the other system components, manages devices, transfers data, and detects I/O completion.

## 1.6 Security and Protection

If a computer system has multiple users and allows the concurrent execution of multiple processes, then access to data must be regulated. For that purpose, mechanisms ensure that files, memory segments, CPU, and other resources can be operated on by only those processes that have gained proper authorization from the operating system. For example, memory-addressing hardware ensures that a process can execute only within its own address space. The timer ensures that no process can gain control of the CPU without eventually relinquishing control. Device-control registers are not accessible to users, so the integrity of the various peripheral devices is protected.

**Protection**, then, is any mechanism for controlling the access of processes or users to the resources defined by a computer system. This mechanism must provide means to specify the controls to be imposed and to enforce the controls.

Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by another subsystem that is malfunctioning. Furthermore, an unprotected resource cannot defend against use (or misuse) by an unauthorized or incompetent user. A protection-oriented system provides a means to distinguish between authorized and unauthorized usage, as we discuss in Chapter 17.

A system can have adequate protection but still be prone to failure and allow inappropriate access. Consider a user whose authentication information (her means of identifying herself to the system) is stolen. Her data could be copied or deleted, even though file and memory protection are working. It is the job of **security** to defend a system from external and internal attacks. Such attacks spread across a huge range and include viruses and worms, denial-of-service attacks (which use all of a system's resources and so keep legitimate users out of the system), identity theft, and theft of service (unauthorized use of a system). Prevention of some of these attacks is considered an operating-system function on some systems, while other systems leave it to policy or additional software. Due to the alarming rise in security incidents, operating-system security features are a fast-growing area of research and implementation. We discuss security in Chapter 16.

Protection and security require the system to be able to distinguish among all its users. Most operating systems maintain a list of user names and associated **user identifier** (**user IDs**). In Windows parlance, this is a **security ID** (**SID**). These numerical IDs are unique, one per user. When a user logs in to the system, the authentication stage determines the appropriate user ID for the user. That user ID is associated with all of the user's processes and threads. When an ID needs to be readable by a user, it is translated back to the user name via the user name list.

In some circumstances, we wish to distinguish among sets of users rather than individual users. For example, the owner of a file on a UNIX system may be allowed to issue all operations on that file, whereas a selected set of users may be allowed only to read the file. To accomplish this, we need to define a group name and the set of users belonging to that group. Group functionality can be implemented as a system-wide list of group names and **group identifier** . A user can be in one or more groups, depending on operating-system design decisions. The user's group IDs are also included in every associated process and thread.

In the course of normal system use, the user ID and group ID for a user are sufficient. However, a user sometimes needs to **escalate privileges** to gain extra permissions for an activity. The user may need access to a device that is restricted, for example. Operating systems provide various methods to allow privilege escalation. On UNIX, for instance, the *setuid* attribute on a program causes that program to run with the user ID of the owner of the file, rather than the current user's ID. The process runs with this **effective UID** until it turns off the extra privileges or terminates.

## 1.7 Virtualization

**Virtualization** is a technology that allows us to abstract the hardware of a single computer (the CPU, memory, disk drives, network interface cards, and so forth) into several different execution environments, thereby creating the illusion that each separate environment is running on its own private computer. These environments can be viewed as different individual operating systems (for example, Windows and UNIX) that may be running at the same time and may interact with each other. A user of a **virtual machine** can switch among the various operating systems in the same way a user can switch among the various processes running concurrently in a single operating system.

Virtualization allows operating systems to run as applications within other operating systems. At first blush, there seems to be little reason for such functionality. But the virtualization industry is vast and growing, which is a testament to its utility and importance.

Broadly speaking, virtualization software is one member of a class that also includes emulation. **Emulation**, which involves simulating computer hardware in software, is typically used when the source CPU type is different from the target CPU type. For example, when Apple switched from the IBM Power CPU to the Intel x86 CPU for its desktop and laptop computers, it included an emulation facility called "Rosetta," which allowed applications compiled for the IBM CPU to run on the Intel CPU. That same concept can be extended to allow an entire operating system written for one platform to run on another. Emulation comes at a heavy price, however. Every machine-level instruction that runs natively on the source system must be translated to the equivalent function on the target system, frequently resulting in several target instructions. If the source and target CPUs have similar performance levels, the emulated code may run much more slowly than the native code.

With virtualization, in contrast, an operating system that is natively compiled for a particular CPU architecture runs within another operating system also native to that CPU. Virtualization first came about on IBM mainframes as a method for multiple users to run tasks concurrently. Running multiple virtual machines allowed (and still allows) many users to run tasks on a system designed for a single user. Later, in response to problems with running multiple Microsoft Windows applications on the Intel x86 CPU, VMware created a new virtualization technology in the form of an application that ran on Windows. That application ran one or more **guest** copies of Windows or other native x86 operating systems, each running its own applications. (See Figure 1.16.)

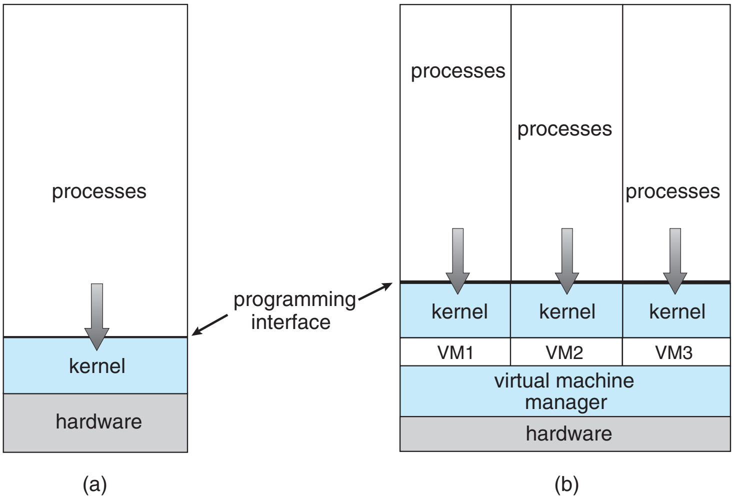


**Figure 1.16** A computer running (a) a single operating system and (b) three virtual machines.

Windows was the **host** operating system, and the VMware application was the **virtual machine manager** (**VMM**). The VMM runs the guest operating systems, manages their resource use, and protects each guest from the others.

Even though modern operating systems are fully capable of running multiple applications reliably, the use of virtualization continues to grow. On laptops and desktops, a VMM allows the user to install multiple operating systems for exploration or to run applications written for operating systems other than the native host. For example, an Apple laptop running macOS on the x86 CPU can run a Windows 10 guest to allow execution of Windows applications. Companies writing software for multiple operating systems can use virtualization to run all of those operating systems on a single physical server for development, testing, and debugging. Within data centers, virtualization has become a common method of executing and managing computing environments. VMMs like VMware ESXand Citrix XenServer no longer run on host operating systems but rather ***are*** the host operating systems, providing services and resource management to virtual machine processes.

With this text, we provide a Linux virtual machine that allows you to run Linux—as well as the development tools we provide—on your personal system regardless of your host operating system. Full details of the features and implementation of virtualization can be found in Chapter 18.

## 1.8 Distributed Systems

A distributed system is a collection of physically separate, possibly heterogeneous computer systems that are networked to provide users with access to the various resources that the system maintains. Access to a shared resource increases computation speed, functionality, data availability, and reliability. Some operating systems generalize network access as a form of file access, with the details of networking contained in the network interface's device driver.

Others make users specifically invoke network functions. Generally, systems contain a mix of the two modes—for example FTP and NFS. The protocols that create a distributed system can greatly affect that system's utility and popularity.

A **network**, in the simplest terms, is a communication path between two or more systems. Distributed systems depend on networking for their functionality. Networks vary by the protocols used, the distances between nodes, and the transport media. **TCP/IP** is the most common network protocol, and it provides the fundamental architecture of the Internet. Most operating systems support TCP/IP, including all general-purpose ones. Some systems support proprietary protocols to suit their needs. For an operating system, it is necessary only that a network protocol have an interface device—a network adapter, for example—with a device driver to manage it, as well as software to handle data. These concepts are discussed throughout this book.

Networks are characterized based on the distances between their nodes. A **local-area network** (**LAN**) connects computers within a room, a building, or a campus. A **wide-area network** (**WAN**) usually links buildings, cities, or countries. A global company may have a WAN to connect its offices worldwide, for example. These networks may run one protocol or several protocols. The continuing advent of new technologies brings about new forms of networks. For example, a **metropolitan-area network** (**MAN**) could link buildings within a city. BlueTooth and 802.11 devices use wireless technology to communicate over a distance of several feet, in essence creating a **personal-area network** (**PAN**) between a phone and a headset or a smartphone and a desktop computer.

The media to carry networks are equally varied. They include copper wires, fiber strands, and wireless transmissions between satellites, microwave dishes, and radios. When computing devices are connected to cellular phones, they create a network. Even very short-range infrared communication can be used for networking. At a rudimentary level, whenever computers communicate, they use or create a network. These networks also vary in their performance and reliability.

Some operating systems have taken the concept of networks and distributed systems further than the notion of providing network connectivity. A **network operating system** is an operating system that provides features such as file sharing across the network, along with a communication scheme that allows different processes on different computers to exchange messages. A computer running a network operating system acts autonomously from all other computers on the network, although it is aware of the network and is able to communicate with other networked computers. A distributed operating system provides a less autonomous environment. The different computers communicate closely enough to provide the illusion that only a single operating system controls the network. We cover computer networks and distributed systems in Chapter 19.

## 1.9 Kernel Data Structures

We turn next to a topic central to operating-system implementation: the way data are structured in the system. In this section, we briefly describe several fundamental data structures used extensively in operating systems. Readers

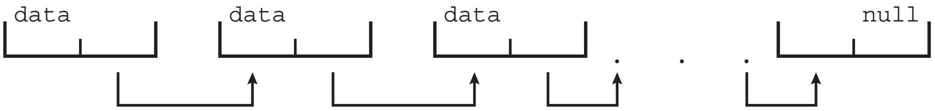


**Figure 1.17** Singly linked list.

who require further details on these structures, as well as others, should consult the bibliography at the end of the chapter.

### 1.9.1 Lists, Stacks, and Queues

An array is a simple data structure in which each element can be accessed directly. For example, main memory is constructed as an array. If the data item being stored is larger than one byte, then multiple bytes can be allocated to the item, and the item is addressed as "item number × item size." But what about storing an item whose size may vary? And what about removing an item if the relative positions of the remaining items must be preserved? In such situations, arrays give way to other data structures.

After arrays, lists are perhaps the most fundamental data structures in computer science. Whereas each item in an array can be accessed directly, the items in a list must be accessed in a particular order. That is, a **list** represents a collection of data values as a sequence. The most common method for implementing this structure is a **linked list**, in which items are linked to one another. Linked lists are of several types:

- In a ***singly linked list,*** each item points to its successor, as illustrated in Figure 1.17.
- In a ***doubly linked list,*** a given item can refer either to its predecessor or to its successor, as illustrated in Figure 1.18.
- In a ***circularly linked list,*** the last element in the list refers to the first element, rather than to null, as illustrated in Figure 1.19.

Linked lists accommodate items of varying sizes and allow easy insertion and deletion of items. One potential disadvantage of using a list is that performance for retrieving a specified item in a list of size $n$ is linear—$O(n)$, as it requires potentially traversing all $n$ elements in the worst case. Lists are sometimes used directly by kernel algorithms. Frequently, though, they are used for constructing more powerful data structures, such as stacks and queues.

A **stack** is a sequentially ordered data structure that uses the last in, first out (**LIFO**) principle for adding and removing items, meaning that the last item

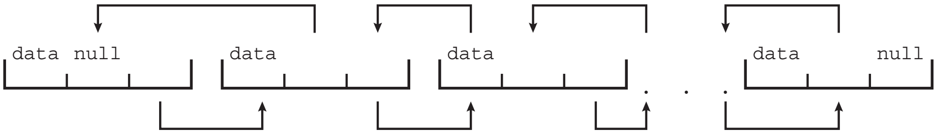


**Figure 1.18** Doubly linked list.

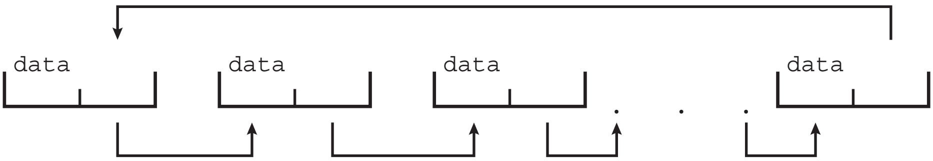


**Figure 1.19** Circularly linked list.

placed onto a stack is the first item removed. The operations for inserting and removing items from a stack are known as ***push*** and ***pop***, respectively. An operating system often uses a stack when invoking function calls. Parameters, local variables, and the return address are pushed onto the stack when a function is called; returning from the function call pops those items off the stack.

A **queue**, in contrast, is a sequentially ordered data structure that uses the first in, first out (**FIFO**) principle: items are removed from a queue in the order in which they were inserted. There are many everyday examples of queues, including shoppers waiting in a checkout line at a store and cars waiting in line at a traffic signal. Queues are also quite common in operating systems—jobs that are sent to a printer are typically printed in the order in which they were submitted, for example. As we shall see in Chapter 5, tasks that are waiting to be run on an available CPU are often organized in queues.

### 1.9.2 Trees

A **tree** is a data structure that can be used to represent data hierarchically. Data values in a tree structure are linked through parent–child relationships. In a **general tree**, a parent may have an unlimited number of children. In a **binary tree**, a parent may have at most two children, which we term the ***left child*** and the ***right child***. A **binary search tree** additionally requires an ordering between the parent's two children in which *left_child* <= *right_child*. Figure 1.20 provides an example of a binary search tree. When we search for an item in a binary search tree, the worst-case performance is $O(n)$ (consider how this can occur). To remedy this situation, we can use an algorithm to create a **balanced binary search tree**. Here, a tree containing $n$ items has at most $lg\ n$ levels, thus ensuring worst-case performance of $O(lg\ n)$. We shall see in Section 5.7.1 that Linux uses a balanced binary search tree (known as a **red-black tree**) as part its CPU-scheduling algorithm.

### 1.9.3 Hash Functions and Maps

A **hash function** takes data as its input, performs a numeric operation on the data, and returns a numeric value. This numeric value can then be used as an index into a table (typically an array) to quickly retrieve the data. Whereas searching for a data item through a list of size $n$ can require up to $O(n)$ comparisons, using a hash function for retrieving data from a table can be as good as $O(1)$, depending on implementation details. Because of this performance, hash functions are used extensively in operating systems.

One potential difficulty with hash functions is that two unique inputs can result in the same output value—that is, they can link to the same table

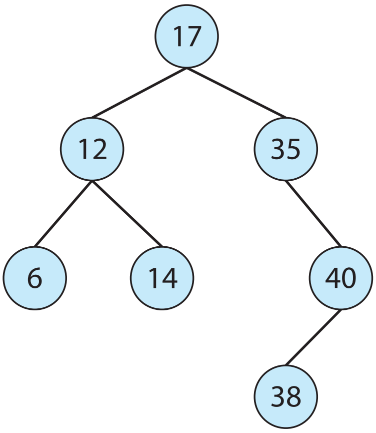


**Figure 1.20** Binary search tree.

location. We can accommodate this ***hash collision*** by having a linked list at the table location that contains all of the items with the same hash value. Of course, the more collisions there are, the less efficient the hash function is.

One use of a hash function is to implement a **hash map**, which associates (or ***maps***) [key:value] pairs using a hash function. Once the mapping is established, we can apply the hash function to the key to obtain the value from the hash map (Figure 1.21). For example, suppose that a user name is mapped to a password. Password authentication then proceeds as follows: a user enters her user name and password. The hash function is applied to the user name, which is then used to retrieve the password. The retrieved password is then compared with the password entered by the user for authentication.

### 1.9.4 Bitmaps

A **bitmap** is a string of $n$ binary digits that can be used to represent the status of $n$ items. For example, suppose we have several resources, and the availability of each resource is indicated by the value of a binary digit: 0 means that the resource is available, while 1 indicates that it is unavailable (or vice versa). The

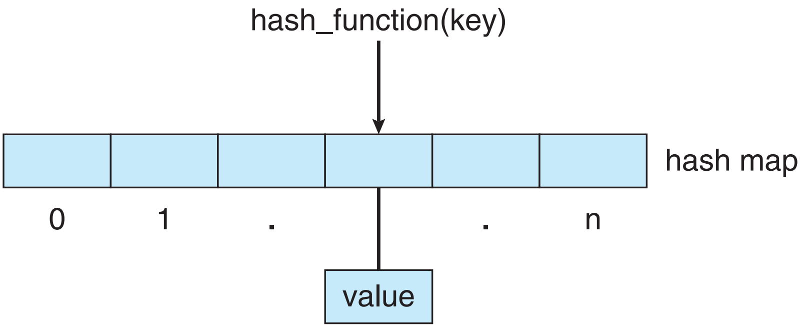


**Figure 1.21** Hash map.

### *LINUX KERNEL DATA STRUCTURES*

The data structures used in the Linux kernel are available in the kernel source code. The ***include*** file `<linux/list.h>` provides details of the linked-list data structure used throughout the kernel. A queue in Linux is known as a `kfifo`, and its implementation can be found in the `kfifo.c` file in the `kernel` directory of the source code. Linux also provides a balanced binary search tree implementation using ***red-black trees***. Details can be found in the include file `<linux/rbtree.h>`.

value of the $i^{th}$ position in the bitmap is associated with the $i^{th}$ resource. As an example, consider the bitmap shown below:

```
001011101
```

Resources 2, 4, 5, 6, and 8 are unavailable; resources 0, 1, 3, and 7 are available.

The power of bitmaps becomes apparent when we consider their space efficiency. If we were to use an eight-bit Boolean value instead of a single bit, the resulting data structure would be eight times larger. Thus, bitmaps are commonly used when there is a need to represent the availability of a large number of resources. Disk drives provide a nice illustration. A medium-sized disk drive might be divided into several thousand individual units, called **disk blocks**. A bitmap can be used to indicate the availability of each disk block.

In summary, data structures are pervasive in operating system implementations. Thus, we will see the structures discussed here, along with others, throughout this text as we explore kernel algorithms and their implementations.

## 1.10 Computing Environments

So far, we have briefly described several aspects of computer systems and the operating systems that manage them. We turn now to a discussion of how operating systems are used in a variety of computing environments.

### 1.10.1 Traditional Computing

As computing has matured, the lines separating many of the traditional computing environments have blurred. Consider the "typical office environment." Just a few years ago, this environment consisted of PCs connected to a network, with servers providing file and print services. Remote access was awkward, and portability was achieved by use of laptop computers.

Today, web technologies and increasing WAN bandwidth are stretching the boundaries of traditional computing. Companies establish **portals**, which provide web accessibility to their internal servers. **Network computers** (or **thin clients**)—which are essentially terminals that understand web-based computing—are used in place of traditional workstations where more security or easier maintenance is desired. Mobile computers can synchronize with PCs to allow very portable use of company information. Mobile devices can also connect to **wireless networks** and cellular data networks to use the company's web portal (as well as the myriad other web resources).

At home, most users once had a single computer with a slow modem connection to the office, the Internet, or both. Today, network-connection speeds once available only at great cost are relatively inexpensive in many places, giving home users more access to more data. These fast data connections are allowing home computers to serve up web pages and to run networks that include printers, client PCs, and servers. Many homes use **firewall** to protect their networks from security breaches. Firewalls limit the communications between devices on a network.

In the latter half of the 20th century, computing resources were relatively scarce. (Before that, they were nonexistent!) For a period of time, systems were either batch or interactive. Batch systems processed jobs in bulk, with predetermined input from files or other data sources. Interactive systems waited for input from users. To optimize the use of the computing resources, multiple users shared time on these systems. These time-sharing systems used a timer and scheduling algorithms to cycle processes rapidly through the CPU, giving each user a share of the resources.

Traditional time-sharing systems are rare today. The same scheduling technique is still in use on desktop computers, laptops, servers, and even mobile computers, but frequently all the processes are owned by the same user (or a single user and the operating system). User processes, and system processes that provide services to the user, are managed so that each frequently gets a slice of computer time. Consider the windows created while a user is working on a PC, for example, and the fact that they may be performing different tasks at the same time. Even a web browser can be composed of multiple processes, one for each website currently being visited, with time sharing applied to each web browser process.

### 1.10.2 Mobile Computing

**Mobile computing** refers to computing on handheld smartphones and tablet computers. These devices share the distinguishing physical features of being portable and lightweight. Historically, compared with desktop and laptop computers, mobile systems gave up screen size, memory capacity, and overall functionality in return for handheld mobile access to services such as e-mail and web browsing. Over the past few years, however, features on mobile devices have become so rich that the distinction in functionality between, say, a consumer laptop and a tablet computer may be difficult to discern. In fact, we might argue that the features of a contemporary mobile device allow it to provide functionality that is either unavailable or impractical on a desktop or laptop computer.

Today, mobile systems are used not only for e-mail and web browsing but also for playing music and video, reading digital books, taking photos, and recording and editing high-definition video. Accordingly, tremendous growth continues in the wide range of applications that run on such devices. Many developers are now designing applications that take advantage of the unique features of mobile devices, such as global positioning system (GPS) chips, accelerometers, and gyroscopes. An embedded GPS chip allows a mobile device to use satellites to determine its precise location on Earth. That functionality is especially useful in designing applications that provide navigation—for example, telling users which way to walk or drive or perhaps directing them to nearby services, such as restaurants. An accelerometer allows a mobile device to detect its orientation with respect to the ground and to detect certain other forces, such as tilting and shaking. In several computer games that employ accelerometers, players interface with the system not by using a mouse or a keyboard but rather by tilting, rotating, and shaking the mobile device! Perhaps more a practical use of these features is found in ***augmented-reality*** applications, which overlay information on a display of the current environment. It is difficult to imagine how equivalent applications could be developed on traditional laptop or desktop computer systems.

To provide access to on-line services, mobile devices typically use either IEEE standard 802.11 wireless or cellular data networks. The memory capacity and processing speed of mobile devices, however, are more limited than those of PCs. Whereas a smartphone or tablet may have 256 GB in storage, it is not uncommon to find 8 TB in storage on a desktop computer. Similarly, because power consumption is such a concern, mobile devices often use processors that are smaller, are slower, and offer fewer processing cores than processors found on traditional desktop and laptop computers.

Two operating systems currently dominate mobile computing: **Apple iOS** and **Google Android**. iOS was designed to run on Apple iPhone and iPad mobile devices. Android powers smartphones and tablet computers available from many manufacturers. We examine these two mobile operating systems in further detail in Chapter 2.

### 1.10.3 Client–Server Computing

Contemporary network architecture features arrangements in which **server systems** satisfy requests generated by **client systems**. This form of specialized distributed system, called a **client–server** system, has the general structure depicted in Figure 1.22.

Server systems can be broadly categorized as compute servers and file servers:

- The **compute-server system** provides an interface to which a client can send a request to perform an action (for example, read data). In response, the server executes the action and sends the results to the client. A server

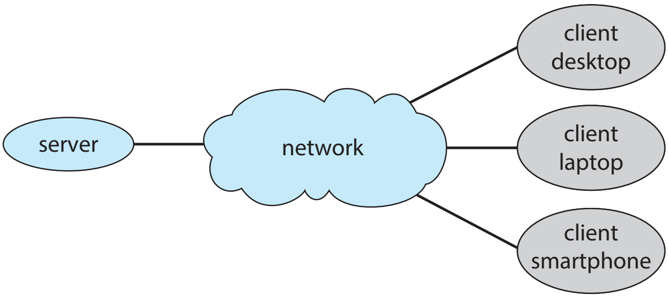


**Figure 1.22** General structure of a client–server system.

running a database that responds to client requests for data is an example of such a system.

- The **file-server system** provides a file-system interface where clients can create, update, read, and delete files. An example of such a system is a web server that delivers files to clients running web browsers. The actual contents of the files can vary greatly, ranging from traditional web pages to rich multimedia content such as high-definition video.

### 1.10.4 Peer-to-Peer Computing

Another structure for a distributed system is the peer-to-peer (P2P) system model. In this model, clients and servers are not distinguished from one another. Instead, all nodes within the system are considered peers, and each may act as either a client or a server, depending on whether it is requesting or providing a service. Peer-to-peer systems offer an advantage over traditional client–server systems. In a client–server system, the server is a bottleneck; but in a peer-to-peer system, services can be provided by several nodes distributed throughout the network.

To participate in a peer-to-peer system, a node must first join the network of peers. Once a node has joined the network, it can begin providing services to—and requesting services from—other nodes in the network. Determining what services are available is accomplished in one of two general ways:

- When a node joins a network, it registers its service with a centralized lookup service on the network. Any node desiring a specific service first contacts this centralized lookup service to determine which node provides the service. The remainder of the communication takes place between the client and the service provider.
- An alternative scheme uses no centralized lookup service. Instead, a peer acting as a client must discover what node provides a desired service by broadcasting a request for the service to all other nodes in the network. The node (or nodes) providing that service responds to the peer making the request. To support this approach, a ***discovery protocol*** must be provided that allows peers to discover services provided by other peers in the network. Figure 1.23 illustrates such a scenario.

Peer-to-peer networks gained widespread popularity in the late 1990s with several file-sharing services, such as Napster and Gnutella, that enabled peers to exchange files with one another. The Napster system used an approach similar to the first type described above: a centralized server maintained an index of all files stored on peer nodes in the Napster network, and the actual exchange of files took place between the peer nodes. The Gnutella system used a technique similar to the second type: a client broadcast file requests to other nodes in the system, and nodes that could service the request responded directly to the client. Peer-to-peer networks can be used to exchange copyrighted materials (music, for example) anonymously, and there are laws governing the distribution of copyrighted material. Notably, Napster ran into legal trouble for copyright infringement, and its services were shut down in 2001. For this reason, the future of exchanging files remains uncertain.

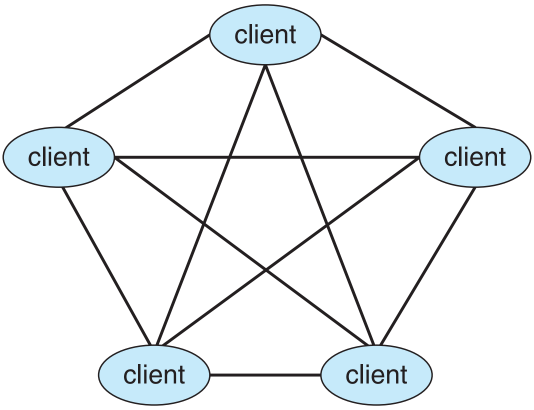


**Figure 1.23** Peer-to-peer system with no centralized service.

Skype is another example of peer-to-peer computing. It allows clients to make voice calls and video calls and to send text messages over the Internet using a technology known as **voice over IP** (**VoIP**). Skype uses a hybrid peer-to-peer approach. It includes a centralized login server, but it also incorporates decentralized peers and allows two peers to communicate.

### 1.10.5 Cloud Computing

**Cloud computing** is a type of computing that delivers computing, storage, and even applications as a service across a network. In some ways, it's a logical extension of virtualization, because it uses virtualization as a base for its functionality. For example, the Amazon Elastic Compute Cloud (**ec2**) facility has thousands of servers, millions of virtual machines, and petabytes of storage available for use by anyone on the Internet. Users pay per month based on how much of those resources they use. There are actually many types of cloud computing, including the following:

- **Public cloud**—a cloud available via the Internet to anyone willing to pay for the services
- **Private cloud**—a cloud run by a company for that company's own use
- **Hybrid cloud**—a cloud that includes both public and private cloud components
- Software as a service (**SaaS**)—one or more applications (such as word processors or spreadsheets) available via the Internet
- Platform as a service (**PaaS**)—a software stack ready for application use via the Internet (for example, a database server)
- Infrastructure as a service (**IaaS**)—servers or storage available over the Internet (for example, storage available for making backup copies of production data)

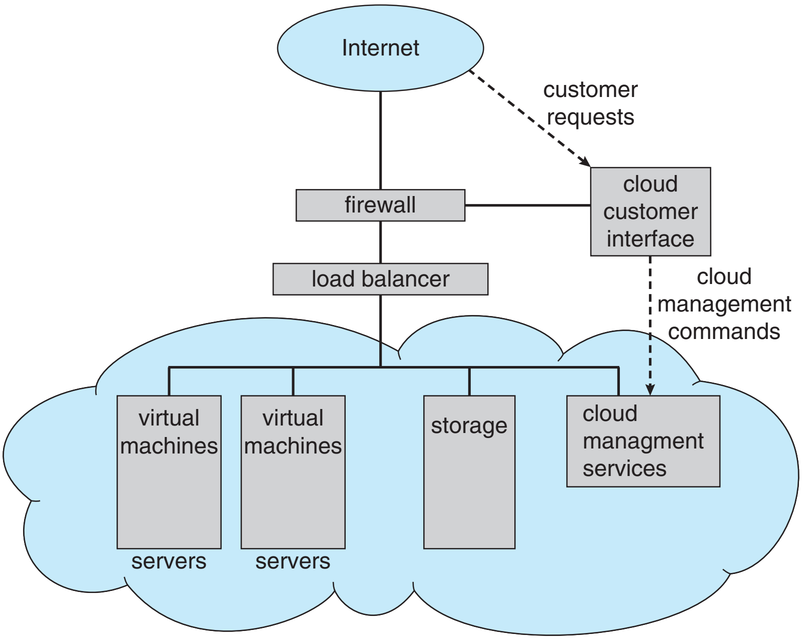


**Figure 1.24** Cloud computing.

These cloud-computing types are not discrete, as a cloud computing environment may provide a combination of several types. For example, an organization may provide both SaaS and IaaS as publicly available services.

Certainly, there are traditional operating systems within many of the types of cloud infrastructure. Beyond those are the VMMs that manage the virtual machines in which the user processes run. At a higher level, the VMMs themselves are managed by cloud management tools, such as VMware vCloud Director and the open-source Eucalyptus toolset. These tools manage the resources within a given cloud and provide interfaces to the cloud components, making a good argument for considering them a new type of operating system.

Figure 1.24 illustrates a public cloud providing IaaS. Notice that both the cloud services and the cloud user interface are protected by a firewall.

### 1.10.6 Real-Time Embedded Systems

Embedded computers are the most prevalent form of computers in existence. These devices are found everywhere, from car engines and manufacturing robots to optical drives and microwave ovens. They tend to have very specific tasks. The systems they run on are usually primitive, and so the operating systems provide limited features. Usually, they have little or no user interface, preferring to spend their time monitoring and managing hardware devices, such as automobile engines and robotic arms.

These embedded systems vary considerably. Some are general-purpose computers, running standard operating systems—such as Linux—with special-purpose applications to implement the functionality. Others are hardware devices with a special-purpose embedded operating system providing just the functionality desired. Yet others are hardware devices with application-specific integrated circuits (**ASICs**) that perform their tasks without an operating system.

The use of embedded systems continues to expand. The power of these devices, both as standalone units and as elements of networks and the web, is sure to increase as well. Even now, entire houses can be computerized, so that a central computer—either a general-purpose computer or an embedded system—can control heating and lighting, alarm systems, and even coffee makers. Web access can enable a home owner to tell the house to heat up before she arrives home. Someday, the refrigerator will be able to notify the grocery store when it notices the milk is gone.

Embedded systems almost always run **real-time operating systems**. A real-time system is used when rigid time requirements have been placed on the operation of a processor or the flow of data; thus, it is often used as a control device in a dedicated application. Sensors bring data to the computer. The computer must analyze the data and possibly adjust controls to modify the sensor inputs. Systems that control scientific experiments, medical imaging systems, industrial control systems, and certain display systems are real-time systems. Some automobile-engine fuel-injection systems, home-appliance controllers, and weapon systems are also real-time systems.

A real-time system has well-defined, fixed time constraints. Processing ***must*** be done within the defined constraints, or the system will fail. For instance, it would not do for a robot arm to be instructed to halt ***after*** it had smashed into the car it was building. A real-time system functions correctly only if it returns the correct result within its time constraints. Contrast this system with a traditional laptop system where it is desirable (but not mandatory) to respond quickly.

In Chapter 5, we consider the scheduling facility needed to implement real-time functionality in an operating system, and in Chapter 20 we describe the real-time components of Linux.

## 1.11 Free and Open-Source Operating Systems

The study of operating systems has been made easier by the availability of a vast number of free software and open-source releases. Both **free operating systems** and **open-source operating systems** are available in source-code format rather than as compiled binary code. Note, though, that free software and open-source software are two different ideas championed by different groups of people (see http://gnu.org/philosophy/open-source-misses-the-point.html/ for a discussion on the topic). Free software (sometimes referred to as ***free/libre software***) not only makes source code available but also is licensed to allow no-cost use, redistribution, and modification. Open-source software does not necessarily offer such licensing. Thus, although all free software is open source, some open-source software is not "free." GNU/Linux is the most famous open-source operating system, with some distributions free and others open source only (http://www.gnu.org/distros/). Microsoft Windows is a well-known example of the opposite **closed-source** approach. Windows is **proprietary** software—Microsoft owns it, restricts its use, and carefully protects its source code. Apple's macOS operating system comprises a hybrid approach. It contains an open-source kernel named Darwin but includes proprietary, closed-source components as well.

Starting with the source code allows the programmer to produce binary code that can be executed on a system. Doing the opposite—**reverse engineering** the source code from the binaries—is quite a lot of work, and useful items such as comments are never recovered. Learning operating systems by examining the source code has other benefits as well. With the source code in hand, a student can modify the operating system and then compile and run the code to try out those changes, which is an excellent learning tool. This text includes projects that involve modifying operating-system source code, while also describing algorithms at a high level to be sure all important operating-system topics are covered. Throughout the text, we provide pointers to examples of open-source code for deeper study.

There are many benefits to open-source operating systems, including a community of interested (and usually unpaid) programmers who contribute to the code by helping to write it, debug it, analyze it, provide support, and suggest changes. Arguably, open-source code is more secure than closed-source code because many more eyes are viewing the code. Certainly, open-source code has bugs, but open-source advocates argue that bugs tend to be found and fixed faster owing to the number of people using and viewing the code. Companies that earn revenue from selling their programs often hesitate to open-source their code, but Red Hat and a myriad of other companies are doing just that and showing that commercial companies benefit, rather than suffer, when they open-source their code. Revenue can be generated through support contracts and the sale of hardware on which the software runs, for example.

### 1.11.1 History

In the early days of modern computing (that is, the 1950s), software generally came with source code. The original hackers (computer enthusiasts) at MIT's Tech Model Railroad Club left their programs in drawers for others to work on. "Homebrew" user groups exchanged code during their meetings. Company-specific user groups, such as Digital Equipment Corporation's DECUS, accepted contributions of source-code programs, collected them onto tapes, and distributed the tapes to interested members. In 1970, Digital's operating systems were distributed as source code with no restrictions or copyright notice.

Computer and software companies eventually sought to limit the use of their software to authorized computers and paying customers. Releasing only the binary files compiled from the source code, rather than the source code itself, helped them to achieve this goal, as well as protecting their code and their ideas from their competitors. Although the Homebrew user groups of the 1970s exchanged code during their meetings, the operating systems for hobbyist machines (such as CPM) were proprietary. By 1980, proprietary software was the usual case.

### 1.11.2 Free Operating Systems

To counter the move to limit software use and redistribution, Richard Stallman in 1984 started developing a free, UNIX-compatible operating system called GNU(which is a recursive acronym for "GNU's Not Unix!"). To Stallman, "free" refers to freedom of use, not price. The free-software movement does not object to trading a copy for an amount of money but holds that users are entitled to four certain freedoms: (1) to freely run the program, (2) to study and change the source code, and to give or sell copies either (3) with or (4) without changes. In 1985, Stallman published the GNU Manifesto, which argues that all software should be free. He also formed the **Free Software Foundation** (**FSF**) with the goal of encouraging the use and development of free software.

The FSF uses the copyrights on its programs to implement "copyleft," a form of licensing invented by Stallman. Copylefting a work gives anyone that possesses a copy of the work the four essential freedoms that make the work free, with the condition that redistribution must preserve these freedoms. The **GNU General Public License** (**GPL**) is a common license under which free software is released. Fundamentally, the GPL requires that the source code be distributed with any binaries and that all copies (including modified versions) be released under the same GPL license. The Creative Commons "Attribution Sharealike" license is also a copyleft license; "sharealike" is another way of stating the idea of copyleft.

### 1.11.3 GNU/Linux

As an example of a free and open-source operating system, consider **GNU/Linux**. By 1991, the GNU operating system was nearly complete. The GNU Project had developed compilers, editors, utilities, libraries, and games — whatever parts it could not find elsewhere. However, the GNU kernel never became ready for prime time. In 1991, a student in Finland, Linus Torvalds, released a rudimentary UNIX-like kernel using the GNU compilers and tools and invited contributions worldwide. The advent of the Internet meant that anyone interested could download the source code, modify it, and submit changes to Torvalds. Releasing updates once a week allowed this so-called "Linux" operating system to grow rapidly, enhanced by several thousand programmers. In 1991, Linux was not free software, as its license permitted only noncommercial redistribution. In 1992, however, Torvalds rereleased Linux under the GPL, making it free software (and also, to use a term coined later, "open source").

The resulting GNU/Linux operating system (with the kernel properly called Linux but the full operating system including GNU tools called GNU/Linux) has spawned hundreds of unique **distributions**, or custom builds, of the system. Major distributions include Red Hat, SUSE, Fedora, Debian, Slackware, and Ubuntu. Distributions vary in function, utility, installed applications, hardware support, user interface, and purpose. For example, Red Hat Enterprise Linux is geared to large commercial use. PCLinuxOS is a **live CD**—an operating system that can be booted and run from a CD-ROM without being installed on a system's boot disk. A variant of PCLinuxOS—called PCLinuxOS Supergamer DVD—is a **live DVD** that includes graphics drivers and games. A gamer can run it on any compatible system simply by booting from the DVD. When the gamer is finished, a reboot of the system resets it to its installed operating system.

You can run Linux on a Windows (or other) system using the following simple, free approach:

1. Download the free Virtualbox VMM tool from

   https://www.virtualbox.org/

   and install it on your system.
2. Choose to install an operating system from scratch, based on an installation image like a CD, or choose pre-built operating-system images that can be installed and run more quickly from a site like

   http://virtualboxes.org/images/

   These images are preinstalled with operating systems and applications and include many flavors of GNU/Linux.
3. Boot the virtual machine within Virtualbox.

An alternative to using Virtualbox is to use the free program Qemu (http://wiki.qemu.org/Download/), which includes the `qemu-img` command for converting Virtualbox images to Qemu images to easily import them.

With this text, we provide a virtual machine image of GNU/Linux running the Ubuntu release. This image contains the GNU/Linux source code as well as tools for software development. We cover examples involving the GNU/Linux image throughout this text, as well as in a detailed case study in Chapter 20.

### 1.11.4 BSD UNIX

**BSD UNIX** has a longer and more complicated history than Linux. It started in 1978 as a derivative of AT&T's UNIX. Releases from the University of California at Berkeley (UCB) came in source and binary form, but they were not open source because a license from AT&T was required. BSD UNIX's development was slowed by a lawsuit by AT&T, but eventually a fully functional, open-source version, 4.4BSD-lite, was released in 1994.

Just as with Linux, there are many distributions of BSD UNIX, including FreeBSD, NetBSD, OpenBSD, and DragonflyBSD. To explore the source code of FreeBSD, simply download the virtual machine image of the version of interest and boot it within Virtualbox, as described above for Linux. The source code comes with the distribution and is stored in `/usr/src/`. The kernel source code is in `/usr/src/sys`. For example, to examine the virtual memory implementation code in the FreeBSD kernel, see the files in `/usr/src/sys/vm`. Alternatively, you can simply view the source code online at https://svnweb.freebsd.org.

As with many open-source projects, this source code is contained in and controlled by a **version control system**—in this case, "subversion" (https://subversion.apache.org/source-code). Version control systems allow a user to "pull" an entire source code tree to his computer and "push" any changes back into the repository for others to then pull. These systems also provide other features, including an entire history of each file and a conflict resolution feature in case the same file is changed concurrently. Another version control system is **git**, which is used for GNU/Linux, as well as other programs (http://www.git-scm.com).

Darwin, the core kernel component of macOS, is based on BSD UNIX and is open-sourced as well. That source code is available from http://www.opensource.apple.com/. Every macOS release has its open-source components posted at that site. The name of the package that contains the kernel begins with "xnu." Apple also provides extensive developer tools, documentation, and support at http://developer.apple.com.

### *THE STUDY OF OPERATING SYSTEMS*

There has never been a more interesting time to study operating systems, and it has never been easier. The open-source movement has overtaken operating systems, causing many of them to be made available in both source and binary (executable) format. The list of operating systems available in both formats includes Linux, BSD UNIX, Solaris, and part of macOS. The availability of source code allows us to study operating systems from the inside out. Questions that we could once answer only by looking at documentation or the behavior of an operating system we can now answer by examining the code itself.

Operating systems that are no longer commercially viable have been open-sourced as well, enabling us to study how systems operated in a time of fewer CPU, memory, and storage resources. An extensive but incomplete list of open-source operating-system projects is available from http://dmoz.org/Computers/Software/Operating_Systems/Open_Source/.

In addition, the rise of virtualization as a mainstream (and frequently free) computer function makes it possible to run many operating systems on top of one core system. For example, VMware (http://www.vmware.com) provides a free "player" for Windows on which hundreds of free "virtual appliances" can run. Virtualbox (http://www.virtualbox.com) provides a free, open-source virtual machine manager on many operating systems. Using such tools, students can try out hundreds of operating systems without dedicated hardware.

In some cases, simulators of specific hardware are also available, allowing the operating system to run on "native" hardware, all within the confines of a modern computer and modern operating system. For example, a DECSYSTEM-20 simulator running on macOS can boot TOPS-20, load the source tapes, and modify and compile a new TOPS-20 kernel. An interested student can search the Internet to find the original papers that describe the operating system, as well as the original manuals.

The advent of open-source operating systems has also made it easier to make the move from student to operating-system developer. With some knowledge, some effort, and an Internet connection, a student can even create a new operating-system distribution. Not so many years ago, it was difficult or impossible to get access to source code. Now, such access is limited only by how much interest, time, and disk space a student has.

### 1.11.5 Solaris

**Solaris** is the commercial UNIX-based operating system of Sun Microsystems. Originally, Sun's **SunOS** operating system was based on BSD UNIX. Sun moved to AT&T's System V UNIX as its base in 1991. In 2005, Sun open-sourced most of the Solaris code as the OpenSolaris project. The purchase of Sun by Oracle in 2009, however, left the state of this project unclear.

Several groups interested in using OpenSolaris have expanded its features, and their working set is Project Illumos, which has expanded from the OpenSolaris base to include more features and to be the basis for several products. Illumos is available at http://wiki.illumos.org.

### 1.11.6 Open-Source Systems as Learning Tools

The free-software movement is driving legions of programmers to create thousands of open-source projects, including operating systems. Sites like http://freshmeat.net/ and http://distrowatch.com/ provide portals to many of these projects. As we stated earlier, open-source projects enable students to use source code as a learning tool. They can modify programs and test them, help find and fix bugs, and otherwise explore mature, full-featured operating systems, compilers, tools, user interfaces, and other types of programs. The availability of source code for historic projects, such as Multics, can help students to understand those projects and to build knowledge that will help in the implementation of new projects.

Another advantage of working with open-source operating systems is their diversity. GNU/Linux and BSD UNIX are both open-source operating systems, for instance, but each has its own goals, utility, licensing, and purpose. Sometimes, licenses are not mutually exclusive and cross-pollination occurs, allowing rapid improvements in operating-system projects. For example, several major components of OpenSolaris have been ported to BSD UNIX. The advantages of free software and open sourcing are likely to increase the number and quality of open-source projects, leading to an increase in the number of individuals and companies that use these projects.

## 1.12 Summary

- An operating system is software that manages the computer hardware, as well as providing an environment for application programs to run.
- Interrupts are a key way in which hardware interacts with the operating system. A hardware device triggers an interrupt by sending a signal to the CPU to alert the CPU that some event requires attention. The interrupt is managed by the interrupt handler.
- For a computer to do its job of executing programs, the programs must be in main memory, which is the only large storage area that the processor can access directly.
- The main memory is usually a volatile storage device that loses its contents when power is turned off or lost.

- Nonvolatile storage is an extension of main memory and is capable of holding large quantities of data permanently.
- The most common nonvolatile storage device is a hard disk, which can provide storage of both programs and data.
- The wide variety of storage systems in a computer system can be organized in a hierarchy according to speed and cost. The higher levels are expensive, but they are fast. As we move down the hierarchy, the cost per bit generally decreases, whereas the access time generally increases.
- Modern computer architectures are multiprocessor systems in which each CPU contains several computing cores.
- To best utilize the CPU, modern operating systems employ multiprogramming, which allows several jobs to be in memory at the same time, thus ensuring that the CPU always has a job to execute.
- Multitasking is an extension of multiprogramming wherein CPU scheduling algorithms rapidly switch between processes, providing users with a fast response time.
- To prevent user programs from interfering with the proper operation of the system, the system hardware has two modes: user mode and kernel mode.
- Various instructions are privileged and can be executed only in kernel mode. Examples include the instruction to switch to kernel mode, I/O control, timer management, and interrupt management.
- A process is the fundamental unit of work in an operating system. Process management includes creating and deleting processes and providing mechanisms for processes to communicate and synchronize with each other.
- An operating system manages memory by keeping track of what parts of memory are being used and by whom. It is also responsible for dynamically allocating and freeing memory space.
- Storage space is managed by the operating system; this includes providing file systems for representing files and directories and managing space on mass-storage devices.
- Operating systems provide mechanisms for protecting and securing the operating system and users. Protection measures control the access of processes or users to the resources made available by the computer system.
- Virtualization involves abstracting a computer’s hardware into several different execution environments.
- Data structures that are used in an operating system include lists, stacks, queues, trees, and maps.
- Computing takes place in a variety of environments, including traditional computing, mobile computing, client–server systems, peer-to-peer systems, cloud computing, and real-time embedded systems.

- Free and open-source operating systems are available in source-code format. Free software is licensed to allow no-cost use, redistribution, and modification. GNU/Linux, FreeBSD, and Solaris are examples of popular open-source systems.

## Practice Exercises

**1.1** What are the three main purposes of an operating system?

**1.2** We have stressed the need for an operating system to make efficient use of the computing hardware. When is it appropriate for the operating system to forsake this principle and to "waste" resources? Why is such a system not really wasteful?

**1.3** What is the main difficulty that a programmer must overcome in writing an operating system for a real-time environment?

**1.4** Keeping in mind the various definitions of ***operating system,*** consider whether the operating system should include applications such as web browsers and mail programs. Argue both that it should and that it should not, and support your answers.

**1.5** How does the distinction between kernel mode and user mode function as a rudimentary form of protection (security)?

**1.6** Which of the following instructions should be privileged?

a. Set value of timer.
b. Read the clock.
c. Clear memory.
d. Issue a trap instruction.
e. Turn off interrupts.
f. Modify entries in device-status table.
g. Switch from user to kernel mode.
h. Access I/O device.

**1.7** Some early computers protected the operating system by placing it in a memory partition that could not be modified by either the user job or the operating system itself. Describe two difficulties that you think could arise with such a scheme.

**1.8** Some CPUs provide for more than two modes of operation. What are two possible uses of these multiple modes?

**1.9** Timers could be used to compute the current time. Provide a short description of how this could be accomplished.

**1.10** Give two reasons why caches are useful. What problems do they solve? What problems do they cause? If a cache can be made as large as the device for which it is caching (for instance, a cache as large as a disk), why not make it that large and eliminate the device?

**1.11** Distinguish between the client–server and peer-to-peer models of distributed systems.

## Further Reading

Many general textbooks cover operating systems, including [Stallings (2017)] and [Tanenbaum (2014)]. [Hennessy and Patterson (2012)] provide coverage of I/O systems and buses and of system architecture in general. [Kurose and Ross (2017)] provides a general overview of computer networks.

[Russinovich et al. (2017)] give an overview of Microsoft Windows and covers considerable technical detail about the system internals and components. [McDougall and Mauro (2007)] cover the internals of the Solaris operating system. The macOS and iOS internals are discussed in [Levin (2013)]. [Levin (2015)] covers the internals of Android. [Love (2010)] provides an overview of the Linux operating system and great detail about data structures used in the Linux kernel. The Free Software Foundation has published its philosophy at http://www.gnu.org/philosophy/free-software-for-freedom.html.

## Bibliography

**[Hennessy and Patterson (2012)]** J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach,* Fifth Edition, Morgan Kaufmann (2012).

**[Kurose and Ross (2017)]** J. Kurose and K. Ross, *Computer Networking—A Top–Down Approach,* Seventh Edition, Addison-Wesley (2017).

**[Levin (2013)]** J. Levin, *Mac OS X and iOS Internals to the Apple's Core*, Wiley (2013).

**[Levin (2015)]** J. Levin, *Android Internals–A Confectioner's Cookbook. Volume I* (2015).

**[Love (2010)]** R. Love, *Linux Kernel Development,* Third Edition, Developer's Library (2010).

**[McDougall and Mauro (2007)]** R. McDougall and J. Mauro, *Solaris Internals,* Second Edition, Prentice Hall (2007).

**[Russinovich et al. (2017)]** M. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals–Part 1,* Seventh Edition, Microsoft Press (2017).

**[Stallings (2017)]** W. Stallings, *Operating Systems, Internals and Design Principles (9th Edition)* Ninth Edition, Prentice Hall (2017).

**[Tanenbaum (2014)]** A. S. Tanenbaum, *Modern Operating Systems*, Prentice Hall (2014).

# Chapter 1 Exercises

**1.12** How do clustered systems differ from multiprocessor systems? What is required for two machines belonging to a cluster to cooperate to provide a highly available service?

**1.13** Consider a computing cluster consisting of two nodes running a database. Describe two ways in which the cluster software can manage access to the data on the disk. Discuss the benefits and disadvantages of each.

**1.14** What is the purpose of interrupts? How does an interrupt differ from a trap? Can traps be generated intentionally by a user program? If so, for what purpose?

**1.15** Explain how the Linux kernel variables `HZ` and `jiffies` can be used to determine the number of seconds the system has been running since it was booted.

**1.16** Direct memory access is used for high-speed I/O devices in order to avoid increasing the CPU's execution load.

a. How does the CPU interface with the device to coordinate the transfer?

b. How does the CPU know when the memory operations are complete?

c. The CPU is allowed to execute other programs while the DMA controller is transferring data. Does this process interfere with the execution of the user programs? If so, describe what forms of interference are caused.

**1.17** Some computer systems do not provide a privileged mode of operation in hardware. Is it possible to construct a secure operating system for these computer systems? Give arguments both that it is and that it is not possible.

**1.18** Many SMP systems have different levels of caches; one level is local to each processing core, and another level is shared among all processing cores. Why are caching systems designed this way?

**1.19** Rank the following storage systems from slowest to fastest:

a. Hard-disk drives

b. Registers

c. Optical disk

d. Main memory

e. Nonvolatile memory

f. Magnetic tapes

g. Cache

**1.20** Consider an SMP system similar to the one shown in Figure 1.8. Illustrate with an example how data residing in memory could in fact have a different value in each of the local caches.

**1.21** Discuss, with examples, how the problem of maintaining coherence of cached data manifests itself in the following processing environments:

a. Single-processor systems
b. Multiprocessor systems
c. Distributed systems

**1.22** Describe a mechanism for enforcing memory protection in order to prevent a program from modifying the memory associated with other programs.

**1.23** Which network configuration—LAN or WAN—would best suit the following environments?

a. A campus student union
b. Several campus locations across a statewide university system
c. A neighborhood

**1.24** Describe some of the challenges of designing operating systems for mobile devices compared with designing operating systems for traditional PCs.

**1.25** What are some advantages of peer-to-peer systems over client–server systems?

**1.26** Describe some distributed applications that would be appropriate for a peer-to-peer system.

**1.27** Identify several advantages and several disadvantages of open-source operating systems. Identify the types of people who would find each aspect to be an advantage or a disadvantage.

# Operating-System Structures

CHAPTER 2

An operating system provides the environment within which programs are executed. Internally, operating systems vary greatly in their makeup, since they are organized along many different lines. The design of a new operating system is a major task. It is important that the goals of the system be well defined before the design begins. These goals form the basis for choices among various algorithms and strategies.

We can view an operating system from several vantage points. One view focuses on the services that the system provides; another, on the interface that it makes available to users and programmers; a third, on its components and their interconnections. In this chapter, we explore all three aspects of operating systems, showing the viewpoints of users, programmers, and operating system designers. We consider what services an operating system provides, how they are provided, how they are debugged, and what the various methodologies are for designing such systems. Finally, we describe how operating systems are created and how a computer starts its operating system.

### CHAPTER OBJECTIVES

- Identify services provided by an operating system.
- Illustrate how system calls are used to provide operating system services.
- Compare and contrast monolithic, layered, microkernel, modular, and hybrid strategies for designing operating systems.
- Illustrate the process for booting an operating system.
- Apply tools for monitoring operating system performance.
- Design and implement kernel modules for interacting with a Linux kernel.

## 2.1 Operating-System Services

An operating system provides an environment for the execution of programs. It makes certain services available to programs and to the users of those programs. The specific services provided, of course, differ from one operating

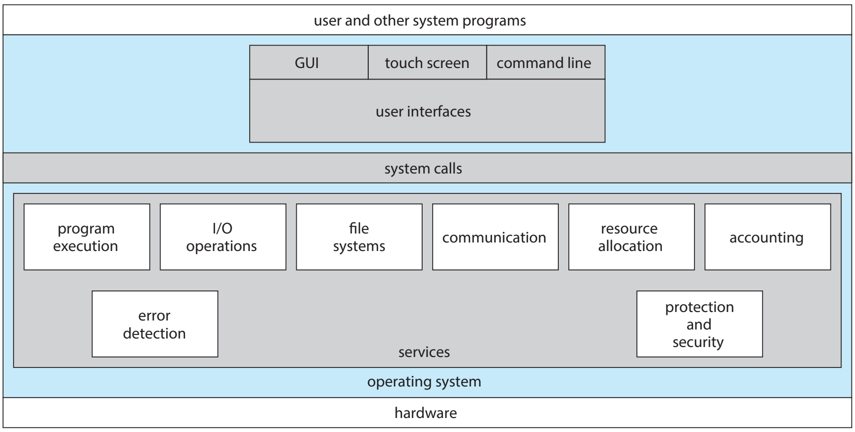

user and other system programs
GUI
touch screen
command line
user interfaces
system calls
program execution
I/O operations
file systems
communication
resource allocation
accounting
error detection
protection and security
services
operating system
hardware

**Figure 2.1** A view of operating system services.

system to another, but we can identify common classes. Figure 2.1 shows one view of the various operating-system services and how they interrelate. Note that these services also make the programming task easier for the programmer.

One set of operating system services provides functions that are helpful to the user.

- **User interface**. Almost all operating systems have a **user interface** (**UI**). This interface can take several forms. Most commonly, a **graphical user interface** (**GUI**) is used. Here, the interface is a window system with a mouse that serves as a pointing device to direct I/O, choose from menus, and make selections and a keyboard to enter text. Mobile systems such as phones and tablets provide a **touch-screen interface**, enabling users to slide their fingers across the screen or press buttons on the screen to select choices. Another option is a **command-line interface** (**CLI**), which uses text commands and a method for entering them (say, a keyboard for typing in commands in a specific format with specific options). Some systems provide two or all three of these variations.
- **Program execution**. The system must be able to load a program into memory and to run that program. The program must be able to end its execution, either normally or abnormally (indicating error).
- **I/O operations**. A running program may require I/O, which may involve a file or an I/O device. For specific devices, special functions may be desired (such as reading from a network interface or writing to a file system). For efficiency and protection, users usually cannot control I/O devices directly. Therefore, the operating system must provide a means to do I/O.
- **File-system manipulation**. The file system is of particular interest. Obviously, programs need to read and write files and directories. They also need to create and delete them by name, search for a given file, and list file information. Finally, some operating systems include permissions management to allow or deny access to files or directories based on file ownership. Many operating systems provide a variety of file systems, sometimes to allow

personal choice and sometimes to provide specific features or performance characteristics.

- **Communications**. There are many circumstances in which one process needs to exchange information with another process. Such communication may occur between processes that are executing on the same computer or between processes that are executing on different computer systems tied together by a network. Communications may be implemented via **shared memory**, in which two or more processes read and write to a shared section of memory, or **message passing**, in which packets of information in predefined formats are moved between processes by the operating system.
- **Error detection**. The operating system needs to be detecting and correcting errors constantly. Errors may occur in the CPU and memory hardware (such as a memory error or a power failure), in I/O devices (such as a parity error on disk, a connection failure on a network, or lack of paper in the printer), and in the user program (such as an arithmetic overflow or an attempt to access an illegal memory location). For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing. Sometimes, it has no choice but to halt the system. At other times, it might terminate an error-causing process or return an error code to a process for the process to detect and possibly correct.

Another set of operating-system functions exists not for helping the user but rather for ensuring the efficient operation of the system itself. Systems with multiple processes can gain efficiency by sharing the computer resources among the different processes.

- **Resource allocation**. When there are multiple processes running at the same time, resources must be allocated to each of them. The operating system manages many different types of resources. Some (such as CPU cycles, main memory, and file storage) may have special allocation code, whereas others (such as I/O devices) may have much more general request and release code. For instance, in determining how best to use the CPU, operating systems have CPU-scheduling routines that take into account the speed of the CPU, the process that must be executed, the number of processing cores on the CPU, and other factors. There may also be routines to allocate printers, USB storage drives, and other peripheral devices.
- **Logging**. We want to keep track of which programs use how much and what kinds of computer resources. This record keeping may be used for accounting (so that users can be billed) or simply for accumulating usage statistics. Usage statistics may be a valuable tool for system administrators who wish to reconfigure the system to improve computing services.
- **Protection and security**. The owners of information stored in a multiuser or networked computer system may want to control use of that information. When several separate processes execute concurrently, it should not be possible for one process to interfere with the others or with the operating system itself. Protection involves ensuring that all access to system resources is controlled. Security of the system from outsiders is also important. Such security starts with requiring each user to authenticate himself

or herself to the system, usually by means of a password, to gain access to system resources. It extends to defending external I/O devices, including network adapters, from invalid access attempts and recording all such connections for detection of break-ins. If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

## 2.2 User and Operating-System Interface

We mentioned earlier that there are several ways for users to interface with the operating system. Here, we discuss three fundamental approaches. One provides a command-line interface, or **command interpreter**, that allows users to directly enter commands to be performed by the operating system. The other two allow users to interface with the operating system via a graphical user interface, or GUI.

### 2.2.1 Command Interpreters

Most operating systems, including Linux, UNIX, and Windows, treat the command interpreter as a special program that is running when a process is initiated or when a user first logs on (on interactive systems). On systems with multiple command interpreters to choose from, the interpreters are known as **shells**. For example, on UNIX and Linux systems, a user may choose among several different shells, including the ***C shell***, ***Bourne-Again shell***, ***Korn shell***, and others. Third-party shells and free user-written shells are also available. Most shells provide similar functionality, and a user's choice of which shell to use is generally based on personal preference. Figure 2.2 shows the Bourne-Again (or `bash`) shell command interpreter being used on macOS.

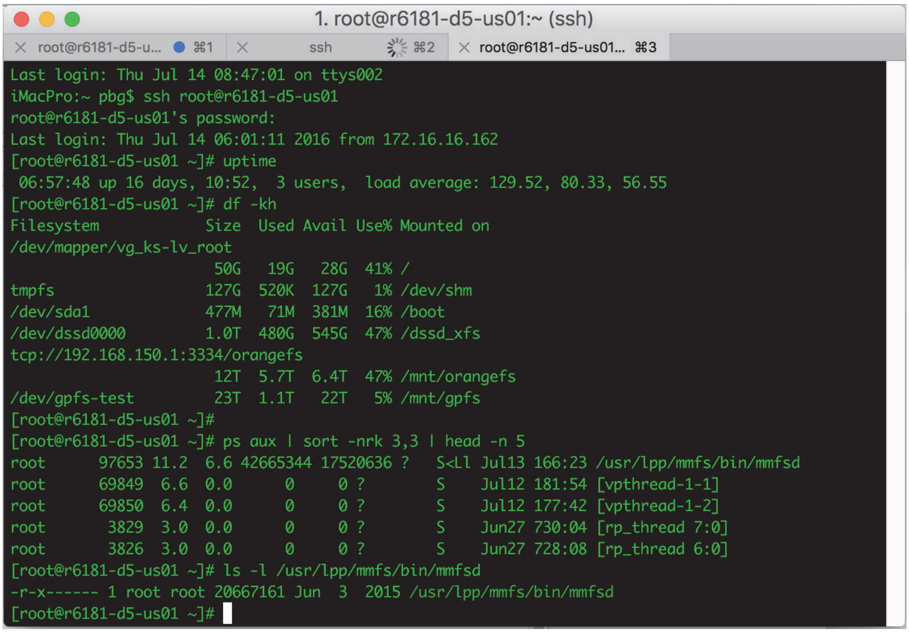
The main function of the command interpreter is to get and execute the next user-specified command. Many of the commands given at this level manipulate files: create, delete, list, print, copy, execute, and so on. The various shells available on UNIX systems operate in this way. These commands can be implemented in two general ways.

In one approach, the command interpreter itself contains the code to execute the command. For example, a command to delete a file may cause the command interpreter to jump to a section of its code that sets up the parameters and makes the appropriate system call. In this case, the number of commands that can be given determines the size of the command interpreter, since each command requires its own implementing code.

An alternative approach—used by UNIX, among other operating systems—implements most commands through system programs. In this case, the command interpreter does not understand the command in any way; it merely uses the command to identify a file to be loaded into memory and executed. Thus, the UNIX command to delete a file

```
rm file.txt
```

would search for a file called `rm`, load the file into memory, and execute it with the parameter `file.txt`. The logic associated with the `rm` command would be



```
Last login: Thu Jul 14 08:47:01 on ttys002
iMacPro:~ pbg$ ssh root@r6181-d5-us01
root@r6181-d5-us01's password:
Last login: Thu Jul 14 06:01:11 2016 from 172.16.16.162
[root@r6181-d5-us01 ~]# uptime
 06:57:48 up 16 days, 10:52,  3 users,  load average: 129.52, 80.33, 56.55
[root@r6181-d5-us01 ~]# df -kh
Filesystem            Size  Used Avail Use% Mounted on
/dev/mapper/vg_ks-lv_root
                       50G   19G   28G  41% /
tmpfs                 127G  520K  127G   1% /dev/shm
/dev/sda1             477M   71M  381M  16% /boot
/dev/dssd0000         1.0T  480G  545G  47% /dssd_xfs
tcp://192.168.150.1:3334/orangefs
                       12T  5.7T  6.4T  47% /mnt/orangefs
/dev/gpfs-test         23T  1.1T   22T   5% /mnt/gpfs
[root@r6181-d5-us01 ~]#
[root@r6181-d5-us01 ~]# ps aux | sort -nrk 3,3 | head -n 5
root      97653 11.2  6.6 42665344 17520636 ?   S<Ll Jul13 166:23 /usr/lpp/mmfs/bin/mmfsd
root      69849  6.6  0.0      0     0 ?        S    Jul12 181:54 [vpthread-1-1]
root      69850  6.4  0.0      0     0 ?        S    Jul12 177:42 [vpthread-1-2]
root       3829  3.0  0.0      0     0 ?        S    Jun27 730:04 [rp_thread 7:0]
root       3826  3.0  0.0      0     0 ?        S    Jun27 728:08 [rp_thread 6:0]
[root@r6181-d5-us01 ~]# ls -l /usr/lpp/mmfs/bin/mmfsd
-r-x------ 1 root root 20667161 Jun  3  2015 /usr/lpp/mmfs/bin/mmfsd
[root@r6181-d5-us01 ~]#
```

**Figure 2.2** The `bash` shell command interpreter in macOS.

defined completely by the code in the file `rm`. In this way, programmers can add new commands to the system easily by creating new files with the proper program logic. The command-interpreter program, which can be small, does not have to be changed for new commands to be added.

### 2.2.2 Graphical User Interface

A second strategy for interfacing with the operating system is through a user-friendly graphical user interface, or GUI. Here, rather than entering commands directly via a command-line interface, users employ a mouse-based window-and-menu system characterized by a **desktop** metaphor. The user moves the mouse to position its pointer on images, or **icons**, on the screen (the desktop) that represent programs, files, directories, and system functions. Depending on the mouse pointer's location, clicking a button on the mouse can invoke a program, select a file or directory—known as a **folder**—or pull down a menu that contains commands.

Graphical user interfaces first appeared due in part to research taking place in the early 1970s at Xerox PARC research facility. The first GUI appeared on the Xerox Alto computer in 1973. However, graphical interfaces became more widespread with the advent of Apple Macintosh computers in the 1980s. The user interface for the Macintosh operating system has undergone various changes over the years, the most significant being the adoption of the *Aqua* interface that appeared with macOS. Microsoft's first version of Windows—Version 1.0—was based on the addition of a GUI interface to the MS-DOS operating system. Later versions of Windows have made significant changes in the appearance of the GUI along with several enhancements in its functionality.

Traditionally, UNIX systems have been dominated by command-line interfaces. Various GUI interfaces are available, however, with significant development in GUI designs from various open-source projects, such as ***K Desktop Environment*** (or ***KDE***) and the ***GNOME*** desktop by the GNU project. Both the KDE and GNOME desktops run on Linux and various UNIX systems and are available under open-source licenses, which means their source code is readily available for reading and for modification under specific license terms.

### 2.2.3 Touch-Screen Interface

Because a either a command-line interface or a mouse-and-keyboard system is impractical for most mobile systems, smartphones and handheld tablet computers typically use a touch-screen interface. Here, users interact by making **gestures** on the touch screen—for example, pressing and swiping fingers across the screen. Although earlier smartphones included a physical keyboard, most smartphones and tablets now simulate a keyboard on the touch screen. Figure 2.3 illustrates the touch screen of the Apple iPhone. Both the iPad and the iPhone use the **Springboard** touch-screen interface.

### 2.2.4 Choice of Interface

The choice of whether to use a command-line or GUI interface is mostly one of personal preference. **System administrators** who manage computers and **power users** who have deep knowledge of a system frequently use the

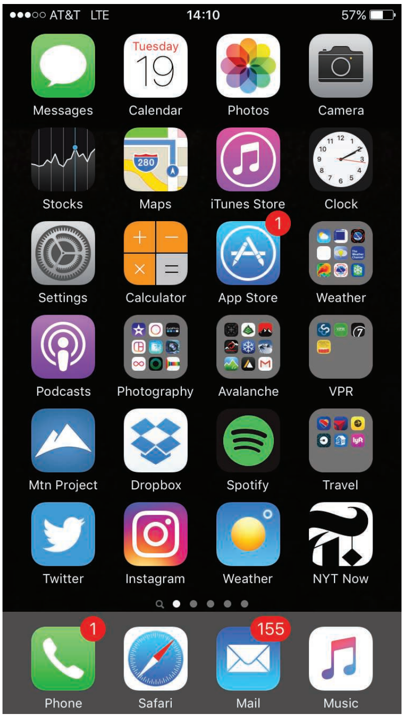

●●●○○ AT&T LTE
14:10
57%
Messages
Calendar
Photos
Camera
Stocks
Maps
iTunes Store
Clock
Settings
Calculator
App Store
Weather
Podcasts
Photography
Avalanche
VPR
Mtn Project
Dropbox
Spotify
Travel
Twitter
Instagram
Weather
NYT Now
Phone
Safari
Mail
Music

**Figure 2.3** The iPhone touch screen.

command-line interface. For them, it is more efficient, giving them faster access to the activities they need to perform. Indeed, on some systems, only a subset of system functions is available via the GUI, leaving the less common tasks to those who are command-line knowledgeable. Further, command-line interfaces usually make repetitive tasks easier, in part because they have their own programmability. For example, if a frequent task requires a set of command-line steps, those steps can be recorded into a file, and that file can be run just like a program. The program is not compiled into executable code but rather is interpreted by the command-line interface. These **shell scripts** are very common on systems that are command-line oriented, such as UNIX and Linux.

In contrast, most Windows users are happy to use the Windows GUI environment and almost never use the shell interface. Recent versions of the Windows operating system provide both a standard GUI for desktop and traditional laptops and a touch screen for tablets. The various changes undergone by the Macintosh operating systems also provide a nice study in contrast. Historically, Mac OS has not provided a command-line interface, always requiring its users to interface with the operating system using its GUI. However, with the release of macOS (which is in part implemented using a UNIX kernel), the operating system now provides both an Aqua GUI and a command-line interface. Figure 2.4 is a screenshot of the macOS GUI.

Although there are apps that provide a command-line interface for iOS and Android mobile systems, they are rarely used. Instead, almost all users of mobile systems interact with their devices using the touch-screen interface.

The user interface can vary from system to system and even from user to user within a system; however, it typically is substantially removed from the actual system structure. The design of a useful and intuitive user interface is therefore not a direct function of the operating system. In this book, we concentrate on the fundamental problems of providing adequate service to

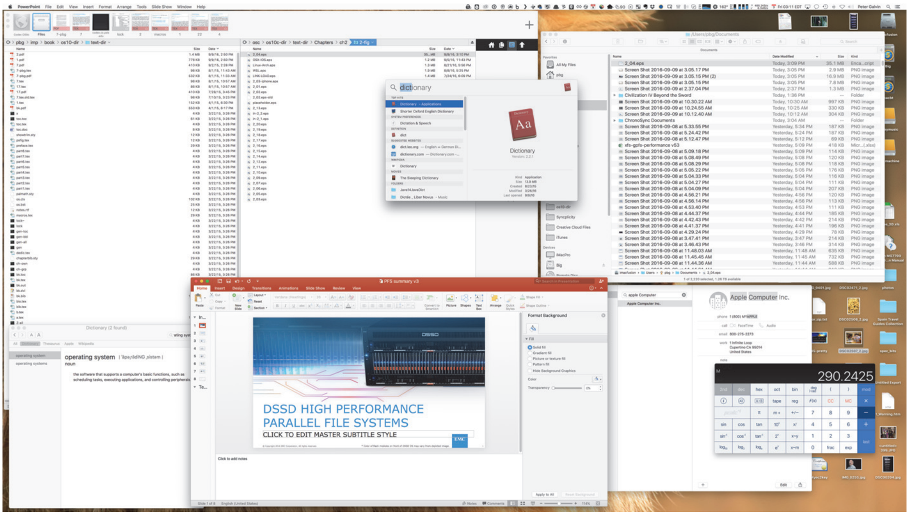


**Figure 2.4** The macOS GUI.

user programs. From the point of view of the operating system, we do not distinguish between user programs and system programs.

## 2.3 System Calls

**System calls** provide an interface to the services made available by an operating system. These calls are generally available as functions written in C and C++, although certain low-level tasks (for example, tasks where hardware must be accessed directly) may have to be written using assembly-language instructions.

### 2.3.1 Example

Before we discuss how an operating system makes system calls available, let's first use an example to illustrate how system calls are used: writing a simple program to read data from one file and copy them to another file. The first input that the program will need is the names of the two files: the input file and the output file. These names can be specified in many ways, depending on the operating-system design. One approach is to pass the names of the two files as part of the command—for example, the UNIX `cp` command:

```
cp in.txt out.txt
```

This command copies the input file `in.txt` to the output file `out.txt`. A second approach is for the program to ask the user for the names. In an interactive system, this approach will require a sequence of system calls, first to write a prompting message on the screen and then to read from the keyboard the characters that define the two files. On mouse-based and icon-based systems, a menu of file names is usually displayed in a window. The user can then use the mouse to select the source name, and a window can be opened for the destination name to be specified. This sequence requires many I/O system calls.

Once the two file names have been obtained, the program must open the input file and create and open the output file. Each of these operations requires another system call. Possible error conditions for each system call must be handled. For example, when the program tries to open the input file, it may find that there is no file of that name or that the file is protected against access. In these cases, the program should output an error message (another sequence of system calls) and then terminate abnormally (another system call). If the input file exists, then we must create a new output file. We may find that there is already an output file with the same name. This situation may cause the program to abort (a system call), or we may delete the existing file (another system call) and create a new one (yet another system call). Another option, in an interactive system, is to ask the user (via a sequence of system calls to output the prompting message and to read the response from the terminal) whether to replace the existing file or to abort the program.

When both files are set up, we enter a loop that reads from the input file (a system call) and writes to the output file (another system call). Each read and write must return status information regarding various possible error conditions. On input, the program may find that the end of the file has been

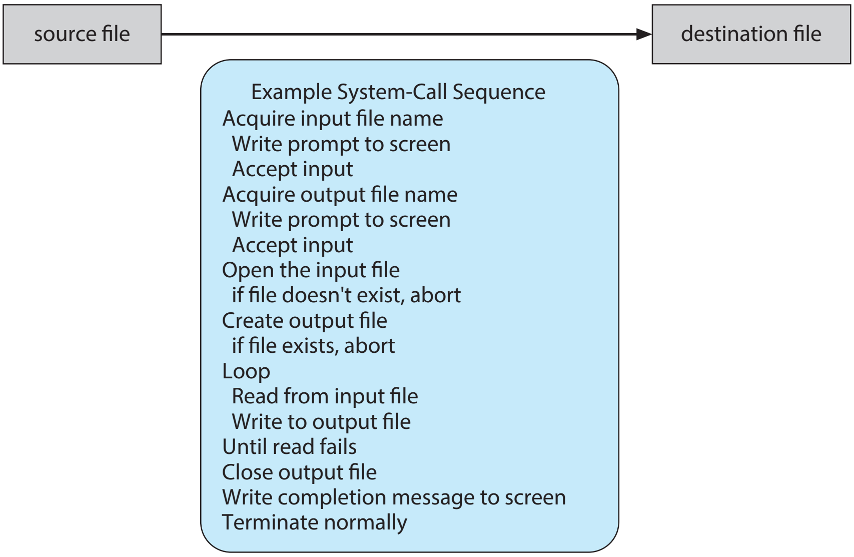

source file
destination file
Example System-Call Sequence
Acquire input file name
Write prompt to screen
Accept input
Acquire output file name
Write prompt to screen
Accept input
Open the input file
if file doesn't exist, abort
Create output file
if file exists, abort
Loop
Read from input file
Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally

**Figure 2.5** Example of how system calls are used.

reached or that there was a hardware failure in the read (such as a parity error). The write operation may encounter various errors, depending on the output device (for example, no more available disk space).

Finally, after the entire file is copied, the program may close both files (two system calls), write a message to the console or window (more system calls), and finally terminate normally (the final system call). This system-call sequence is shown in Figure 2.5.

### 2.3.2 Application Programming Interface

As you can see, even simple programs may make heavy use of the operating system. Frequently, systems execute thousands of system calls per second. Most programmers never see this level of detail, however. Typically, application developers design programs according to an **application programming interface** (**API**). The API specifies a set of functions that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect. Three of the most common APIs available to application programmers are the Windows API for Windows systems, the POSIX API for POSIX-based systems (which include virtually all versions of UNIX, Linux, and macOS), and the Java API for programs that run on the Java virtual machine. A programmer accesses an API via a library of code provided by the operating system. In the case of UNIX and Linux for programs written in the C language, the library is called **libc**. Note that—unless specified—the system-call names used throughout this text are generic examples. Each operating system has its own name for each system call.

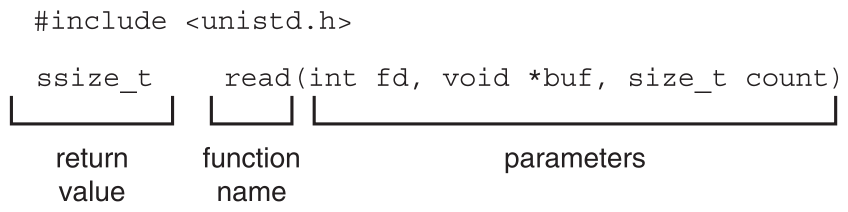
Behind the scenes, the functions that make up an API typically invoke the actual system calls on behalf of the application programmer. For example, the Windows function `CreateProcess()` (which, unsurprisingly, is used to create

### *EXAMPLE OF STANDARD API*

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:



```
#include <unistd.h>

ssize_t    read(int fd, void *buf, size_t count)
```

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer into which the data will be read
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

a new process) actually invokes the `NTCreateProcess()` system call in the Windows kernel.

Why would an application programmer prefer programming according to an API rather than invoking actual system calls? There are several reasons for doing so. One benefit concerns program portability. An application programmer designing a program using an API can expect her program to compile and run on any system that supports the same API (although, in reality, architectural differences often make this more difficult than it may appear). Furthermore, actual system calls can often be more detailed and difficult to work with than the API available to an application programmer. Nevertheless, there often exists a strong correlation between a function in the API and its associated system call within the kernel. In fact, many of the POSIX and Windows APIs are similar to the native system calls provided by the UNIX, Linux, and Windows operating systems.

Another important factor in handling system calls is the **run-time environment** (**RTE**)—the full suite of software needed to execute applications written in a given programming language, including its compilers or interpreters as well as other software, such as libraries and loaders. The RTE provides a

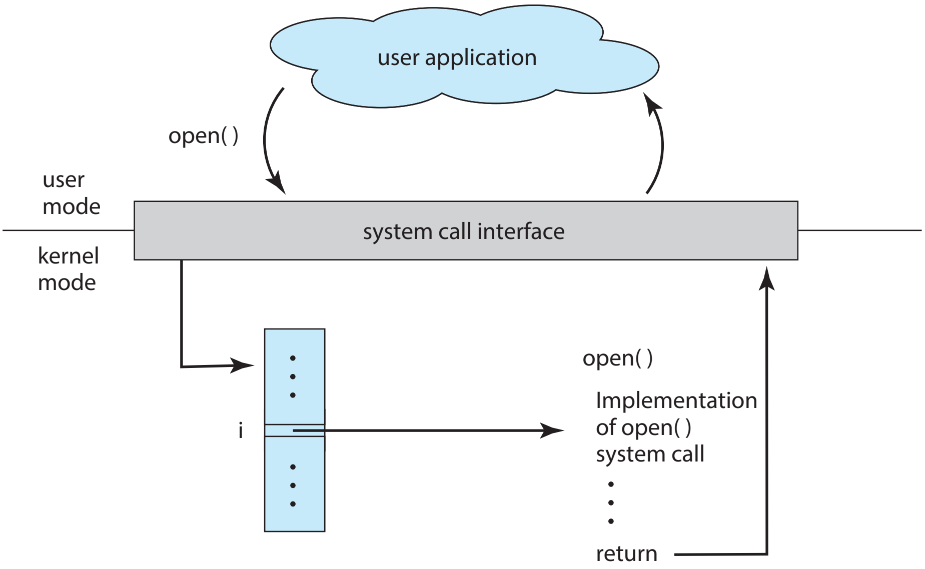


**Figure 2.6** The handling of a user application invoking the `open()` system call.

**system-call interface** that serves as the link to system calls made available by the operating system. The system-call interface intercepts function calls in the API and invokes the necessary system calls within the operating system. Typically, a number is associated with each system call, and the system-call interface maintains a table indexed according to these numbers. The system-call interface then invokes the intended system call in the operating-system kernel and returns the status of the system call.

The caller need know nothing about how the system call is implemented or what it does during execution. Rather, the caller need only obey the API and understand what the operating system will do as a result of the execution of that system call. Thus, most of the details of the operating-system interface are hidden from the programmer by the API and are managed by the RTE. The relationship among an API, the system-call interface, and the operating system is shown in Figure 2.6, which illustrates how the operating system handles a user application invoking the `open()` system call.

System calls occur in different ways, depending on the computer in use. Often, more information is required than simply the identity of the desired system call. The exact type and amount of information vary according to the particular operating system and call. For example, to get input, we may need to specify the file or device to use as the source, as well as the address and length of the memory buffer into which the input should be read. Of course, the device or file and length may be implicit in the call.

Three general methods are used to pass parameters to the operating system. The simplest approach is to pass the parameters in registers. In some cases, however, there may be more parameters than registers. In these cases, the parameters are generally stored in a block, or table, in memory, and the address of the block is passed as a parameter in a register (Figure 2.7). Linux uses a combination of these approaches. If there are five or fewer parameters,

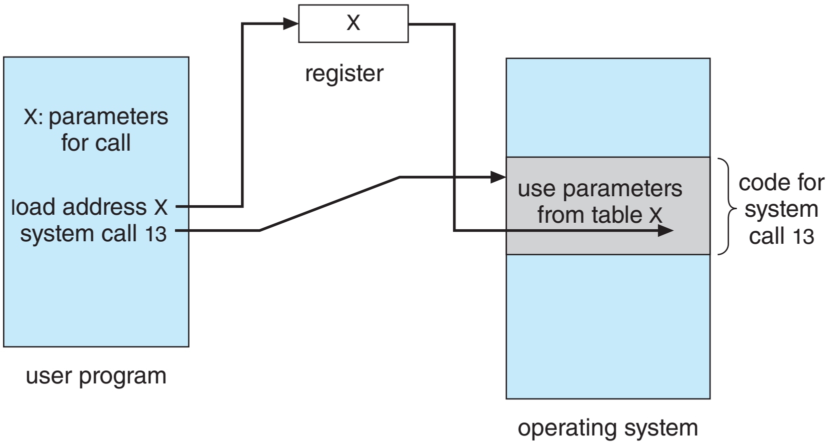


**Figure 2.7** Passing of parameters as a table.

registers are used. If there are more than five parameters, the block method is used. Parameters also can be placed, or **pushed**, onto a **stack** by the program and **popped** off the stack by the operating system. Some operating systems prefer the block or stack method because those approaches do not limit the number or length of parameters being passed.

### 2.3.3 Types of System Calls

System calls can be grouped roughly into six major categories: **process control**, **file management**, **device management**, **information maintenance**, **communications**, and **protection**. Below, we briefly discuss the types of system calls that may be provided by an operating system. Most of these system calls support, or are supported by, concepts and functions that are discussed in later chapters. Figure 2.8 summarizes the types of system calls normally provided by an operating system. As mentioned, in this text, we normally refer to the system calls by generic names. Throughout the text, however, we provide examples of the actual counterparts to the system calls for UNIX, Linux, and Windows systems.

#### 2.3.3.1 Process Control

A running program needs to be able to halt its execution either normally (`end()`) or abnormally (`abort()`). If a system call is made to terminate the currently running program abnormally, or if the program runs into a problem and causes an error trap, a dump of memory is sometimes taken and an error message generated. The dump is written to a special log file on disk and may be examined by a **debugger**—a system program designed to aid the programmer in finding and correcting errors, or **bugs**—to determine the cause of the problem. Under either normal or abnormal circumstances, the operating system must transfer control to the invoking command interpreter. The command interpreter then reads the next command. In an interactive system, the command interpreter simply continues with the next command; it is assumed that the user will issue an appropriate command to respond to

- Process control
    - create process, terminate process
    - load, execute
    - get process attributes, set process attributes
    - wait event, signal event
    - allocate and free memory
- File management
    - create file, delete file
    - open, close
    - read, write, reposition
    - get file attributes, set file attributes
- Device management
    - request device, release device
    - read, write, reposition
    - get device attributes, set device attributes
    - logically attach or detach devices
- Information maintenance
    - get time or date, set time or date
    - get system data, set system data
    - get process, file, or device attributes
    - set process, file, or device attributes
- Communications
    - create, delete communication connection
    - send, receive messages
    - transfer status information
    - attach or detach remote devices
- Protection
    - get file permissions
    - set file permissions

**Figure 2.8** Types of system calls.

### *EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS*

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

| | **Windows** | **Unix** |
|---|---|---|
| **Process control** | `CreateProcess()` | `fork()` |
| | `ExitProcess()` | `exit()` |
| | `WaitForSingleObject()` | `wait()` |
| **File management** | `CreateFile()` | `open()` |
| | `ReadFile()` | `read()` |
| | `WriteFile()` | `write()` |
| | `CloseHandle()` | `close()` |
| **Device management** | `SetConsoleMode()` | `ioctl()` |
| | `ReadConsole()` | `read()` |
| | `WriteConsole()` | `write()` |
| **Information maintenance** | `GetCurrentProcessID()` | `getpid()` |
| | `SetTimer()` | `alarm()` |
| | `Sleep()` | `sleep()` |
| **Communications** | `CreatePipe()` | `pipe()` |
| | `CreateFileMapping()` | `shm_open()` |
| | `MapViewOfFile()` | `mmap()` |
| **Protection** | `SetFileSecurity()` | `chmod()` |
| | `InitlializeSecurityDescriptor()` | `umask()` |
| | `SetSecurityDescriptorGroup()` | `chown()` |

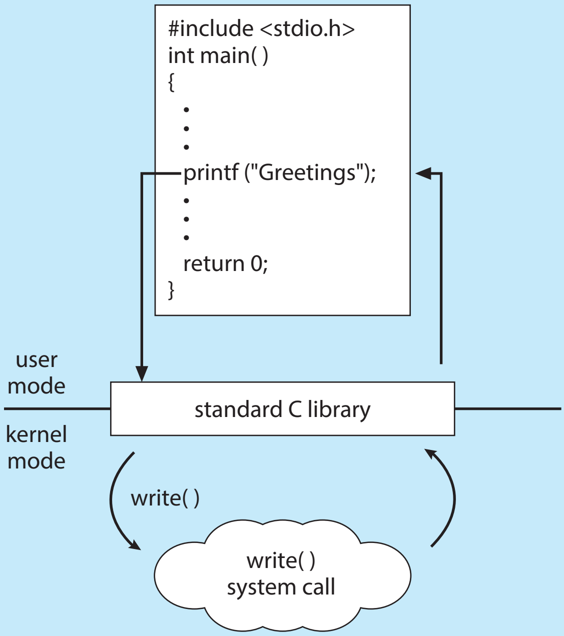
any error. In a GUI system, a pop-up window might alert the user to the error and ask for guidance. Some systems may allow for special recovery actions in case an error occurs. If the program discovers an error in its input and wants to terminate abnormally, it may also want to define an error level. More severe errors can be indicated by a higher-level error parameter. It is then possible to combine normal and abnormal termination by defining a normal termination as an error at level 0. The command interpreter or a following program can use this error level to determine the next action automatically.

A process executing one program may want to `load()` and `execute()` another program. This feature allows the command interpreter to execute a program as directed by, for example, a user command or the click of a mouse. An interesting question is where to return control when the loaded program terminates. This question is related to whether the existing program is lost, saved, or allowed to continue execution concurrently with the new program.

If control returns to the existing program when the new program terminates, we must save the memory image of the existing program; thus, we have

### *THE STANDARD C LIBRARY*

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program:



effectively created a mechanism for one program to call another program. If both programs continue concurrently, we have created a new process to be multiprogrammed. Often, there is a system call specifically for this purpose (`create_process()`).

If we create a new process, or perhaps even a set of processes, we should be able to control its execution. This control requires the ability to determine and reset the attributes of a process, including the process's priority, its maximum allowable execution time, and so on (`get_process_attributes()` and `set_process_attributes()`). We may also want to terminate a process that we created (`terminate_process()`) if we find that it is incorrect or is no longer needed.

Having created new processes, we may need to wait for them to finish their execution. We may want to wait for a certain amount of time to pass (`wait_time()`). More probably, we will want to wait for a specific event to occur (`wait_event()`). The processes should then signal when that event has occurred (`signal_event()`).

Quite often, two or more processes may share data. To ensure the integrity of the data being shared, operating systems often provide system calls allowing

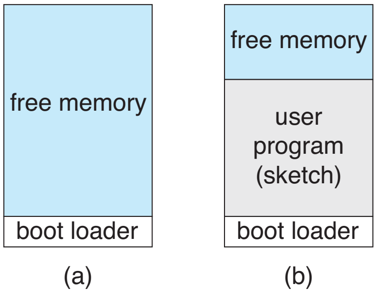


**Figure 2.9** Arduino execution. (a) At system startup. (b) Running a sketch.

a process to **lock** shared data. Then, no other process can access the data until the lock is released. Typically, such system calls include `acquire_lock()` and `release_lock()`. System calls of these types, dealing with the coordination of concurrent processes, are discussed in great detail in Chapter 6 and Chapter 7.

There are so many facets of and variations in process control that we next use two examples—one involving a single-tasking system and the other a multitasking system—to clarify these concepts. The Arduino is a simple hardware platform consisting of a microcontroller along with input sensors that respond to a variety of events, such as changes to light, temperature, and barometric pressure, to just name a few. To write a program for the Arduino, we first write the program on a PC and then upload the compiled program (known as a **sketch**) from the PC to the Arduino's flash memory via a USB connection. The standard Arduino platform does not provide an operating system; instead, a small piece of software known as a **boot loader** loads the sketch into a specific region in the Arduino's memory (Figure 2.9). Once the sketch has been loaded, it begins running, waiting for the events that it is programmed to respond to. For example, if the Arduino's temperature sensor detects that the temperature has exceeded a certain threshold, the sketch may have the Arduino start the motor for a fan. An Arduino is considered a single-tasking system, as only one sketch can be present in memory at a time; if another sketch is loaded, it replaces the existing sketch. Furthermore, the Arduino provides no user interface beyond hardware input sensors.

FreeBSD (derived from Berkeley UNIX) is an example of a multitasking system. When a user logs on to the system, the shell of the user's choice is run, awaiting commands and running programs the user requests. However, since FreeBSD is a multitasking system, the command interpreter may continue running while another program is executed (Figure 2.10). To start a new process, the shell executes a `fork()` system call. Then, the selected program is loaded into memory via an `exec()` system call, and the program is executed. Depending on how the command was issued, the shell then either waits for the process to finish or runs the process "in the background." In the latter case, the shell immediately waits for another command to be entered. When a process is running in the background, it cannot receive input directly from the keyboard, because the shell is using this resource. I/O is therefore done through files or through a GUI interface. Meanwhile, the user is free to ask the shell to run other programs, to monitor the progress of the running process, to change that program's priority, and so on. When the process is done, it executes an `exit()`

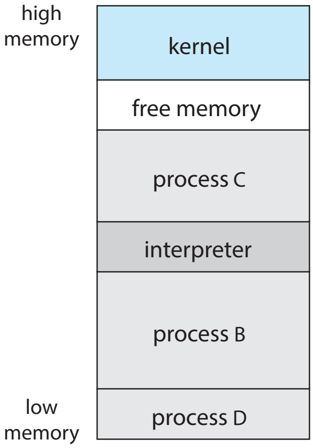


**Figure 2.10** FreeBSD running multiple programs.

system call to terminate, returning to the invoking process a status code of 0 or a nonzero error code. This status or error code is then available to the shell or other programs. Processes are discussed in Chapter 3 with a program example using the `fork()` and `exec()` system calls.

#### 2.3.3.2 File Management

The file system is discussed in more detail in Chapter 13 through Chapter 15. Here, we identify several common system calls dealing with files.

We first need to be able to `create()` and `delete()` files. Either system call requires the name of the file and perhaps some of the file's attributes. Once the file is created, we need to `open()` it and to use it. We may also `read()`, `write()`, or `reposition()` (rewind or skip to the end of the file, for example). Finally, we need to `close()` the file, indicating that we are no longer using it.

We may need these same sets of operations for directories if we have a directory structure for organizing files in the file system. In addition, for either files or directories, we need to be able to determine the values of various attributes and perhaps to set them if necessary. File attributes include the file name, file type, protection codes, accounting information, and so on. At least two system calls, `get_file_attributes()` and `set_file_attributes()`, are required for this function. Some operating systems provide many more calls, such as calls for file `move()` and `copy()`. Others might provide an API that performs those operations using code and other system calls, and others might provide system programs to perform the tasks. If the system programs are callable by other programs, then each can be considered an API by other system programs.

#### 2.3.3.3 Device Management

A process may need several resources to execute—main memory, disk drives, access to files, and so on. If the resources are available, they can be granted, and control can be returned to the user process. Otherwise, the process will have to wait until sufficient resources are available.

The various resources controlled by the operating system can be thought of as devices. Some of these devices are physical devices (for example, disk drives), while others can be thought of as abstract or virtual devices (for example, files). A system with multiple users may require us to first `request()` a device, to ensure exclusive use of it. After we are finished with the device, we `release()` it. These functions are similar to the `open()` and `close()` system calls for files. Other operating systems allow unmanaged access to devices. The hazard then is the potential for device contention and perhaps deadlock, which are described in Chapter 8.

Once the device has been requested (and allocated to us), we can `read()`, `write()`, and (possibly) `reposition()` the device, just as we can with files. In fact, the similarity between I/O devices and files is so great that many operating systems, including UNIX, merge the two into a combined file–device structure. In this case, a set of system calls is used on both files and devices. Sometimes, I/O devices are identified by special file names, directory placement, or file attributes.

The user interface can also make files and devices appear to be similar, even though the underlying system calls are dissimilar. This is another example of the many design decisions that go into building an operating system and user interface.

#### 2.3.3.4 Information Maintenance

Many system calls exist simply for the purpose of transferring information between the user program and the operating system. For example, most systems have a system call to return the current `time()` and `date()`. Other system calls may return information about the system, such as the version number of the operating system, the amount of free memory or disk space, and so on.

Another set of system calls is helpful in debugging a program. Many systems provide system calls to `dump()` memory. This provision is useful for debugging. The program `strace`, which is available on Linux systems, lists each system call as it is executed. Even microprocessors provide a CPU mode, known as **single step**, in which a trap is executed by the CPU after every instruction. The trap is usually caught by a debugger.

Many operating systems provide a time profile of a program to indicate the amount of time that the program executes at a particular location or set of locations. A time profile requires either a tracing facility or regular timer interrupts. At every occurrence of the timer interrupt, the value of the program counter is recorded. With sufficiently frequent timer interrupts, a statistical picture of the time spent on various parts of the program can be obtained.

In addition, the operating system keeps information about all its processes, and system calls are used to access this information. Generally, calls are also used to get and set the process information (`get_process_attributes()` and `set_process_attributes()`). In Section 3.1.3, we discuss what information is normally kept.

#### 2.3.3.5 Communication

There are two common models of interprocess communication: the message-passing model and the shared-memory model. In the **message-passing model**, the communicating processes exchange messages with one another to transfer information. Messages can be exchanged between the processes either directly or indirectly through a common mailbox. Before communication can take place, a connection must be opened. The name of the other communicator must be known, be it another process on the same system or a process on another computer connected by a communications network. Each computer in a network has a **host name** by which it is commonly known. A host also has a network identifier, such as an IP address. Similarly, each process has a **process name**, and this name is translated into an identifier by which the operating system can refer to the process. The `get_hostid()` and `get_processid()` system calls do this translation. The identifiers are then passed to the general-purpose `open()` and `close()` calls provided by the file system or to specific `open_connection()` and `close_connection()` system calls, depending on the system's model of communication. The recipient process usually must give its permission for communication to take place with an `accept_connection()` call. Most processes that will be receiving connections are special-purpose **daemons**, which are system programs provided for that purpose. They execute a `wait_for_connection()` call and are awakened when a connection is made. The source of the communication, known as the **client**, and the receiving daemon, known as a **server**, then exchange messages by using `read_message()` and `write_message()` system calls. The `close_connection()` call terminates the communication.

In the **shared-memory model**, processes use `shared_memory_create()` and `shared_memory_attach()` system calls to create and gain access to regions of memory owned by other processes. Recall that, normally, the operating system tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas. The form of the data is determined by the processes and is not under the operating system's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously. Such mechanisms are discussed in Chapter 6. In Chapter 4, we look at a variation of the process scheme—threads—in which some memory is shared by default.

Both of the models just discussed are common in operating systems, and most systems implement both. Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. It is also easier to implement than is shared memory for intercomputer communication. Shared memory allows maximum speed and convenience of communication, since it can be done at memory transfer speeds when it takes place within a computer. Problems exist, however, in the areas of protection and synchronization between the processes sharing memory.

#### 2.3.3.6 Protection

Protection provides a mechanism for controlling access to the resources provided by a computer system. Historically, protection was a concern only on multiprogrammed computer systems with several users. However, with the advent of networking and the Internet, all computer systems, from servers to mobile handheld devices, must be concerned with protection.

Typically, system calls providing protection include `set_permission()` and `get_permission()`, which manipulate the permission settings of resources such as files and disks. The `allow_user()` and `deny_user()` system calls specify whether particular users can—or cannot—be allowed access to certain resources. We cover protection in Chapter 17 and the much larger issue of security—which involves using protection against external threats—in Chapter 16.

## 2.4 System Services

Another aspect of a modern system is its collection of system services. Recall Figure 1.1, which depicted the logical computer hierarchy. At the lowest level is hardware. Next is the operating system, then the system services, and finally the application programs. **System services**, also known as **system utilities**, provide a convenient environment for program development and execution. Some of them are simply user interfaces to system calls. Others are considerably more complex. They can be divided into these categories:

- **File management**. These programs create, delete, copy, rename, print, list, and generally access and manipulate files and directories.
- **Status information**. Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information. Others are more complex, providing detailed performance, logging, and debugging information. Typically, these programs format and print the output to the terminal or other output devices or files or display it in a window of the GUI. Some systems also support a **registry**, which is used to store and retrieve configuration information.
- **File modification**. Several text editors may be available to create and modify the content of files stored on disk or other storage devices. There may also be special commands to search contents of files or perform transformations of the text.
- **Programming-language support**. Compilers, assemblers, debuggers, and interpreters for common programming languages (such as C, C++, Java, and Python) are often provided with the operating system or available as a separate download.
- **Program loading and execution**. Once a program is assembled or compiled, it must be loaded into memory to be executed. The system may provide absolute loaders, relocatable loaders, linkage editors, and overlay loaders. Debugging systems for either higher-level languages or machine language are needed as well.
- **Communications**. These programs provide the mechanism for creating virtual connections among processes, users, and computer systems. They allow users to send messages to one another's screens, to browse web pages, to send e-mail messages, to log in remotely, or to transfer files from one machine to another.
- **Background services**. All general-purpose systems have methods for launching certain system-program processes at boot time. Some of these processes terminate after completing their tasks, while others continue to

run until the system is halted. Constantly running system-program processes are known as **services**, **subsystems**, or daemons. One example is the network daemon discussed in Section 2.3.3.5. In that example, a system needed a service to listen for network connections in order to connect those requests to the correct processes. Other examples include process schedulers that start processes according to a specified schedule, system error monitoring services, and print servers. Typical systems have dozens of daemons. In addition, operating systems that run important activities in user context rather than in kernel context may use daemons to run these activities.

Along with system programs, most operating systems are supplied with programs that are useful in solving common problems or performing common operations. Such **application programs** include web browsers, word processors and text formatters, spreadsheets, database systems, compilers, plotting and statistical-analysis packages, and games.

The view of the operating system seen by most users is defined by the application and system programs, rather than by the actual system calls. Consider a user's PC. When a user's computer is running the macOS operating system, the user might see the GUI, featuring a mouse-and-windows interface. Alternatively, or even in one of the windows, the user might have a command-line UNIX shell. Both use the same set of system calls, but the system calls look different and act in different ways. Further confusing the user view, consider the user dual-booting from macOS into Windows. Now the same user on the same hardware has two entirely different interfaces and two sets of applications using the same physical resources. On the same hardware, then, a user can be exposed to multiple user interfaces sequentially or concurrently.

## 2.5 Linkers and Loaders

Usually, a program resides on disk as a binary executable file—for example, `a.out` or `prog.exe`. To run on a CPU, the program must be brought into memory and placed in the context of a process. In this section, we describe the steps in this procedure, from compiling a program to placing it in memory, where it becomes eligible to run on an available CPU core. The steps are highlighted in Figure 2.11.

Source files are compiled into object files that are designed to be loaded into any physical memory location, a format known as an **relocatable object file**. Next, the **linker** combines these relocatable object files into a single binary **executable** file. During the linking phase, other object files or libraries may be included as well, such as the standard C or math library (specified with the flag `-lm`).

A **loader** is used to load the binary executable file into memory, where it is eligible to run on a CPU core. An activity associated with linking and loading is **relocation**, which assigns final addresses to the program parts and adjusts code and data in the program to match those addresses so that, for example, the code can call library functions and access its variables as it executes. In Figure 2.11, we see that to run the loader, all that is necessary is to enter the name of the executable file on the command line. When a program name is entered on the

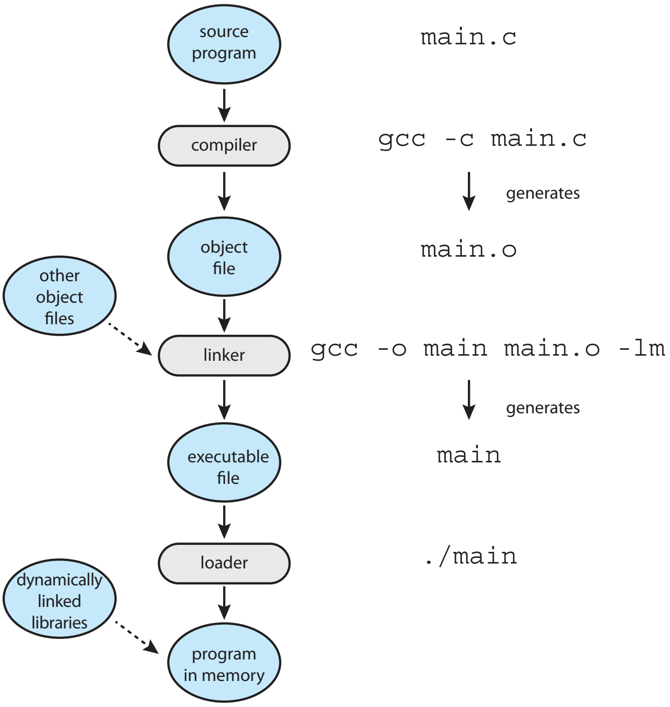

source program
compiler
object file
other object files
linker
executable file
loader
dynamically linked libraries
program in memory
main.c
gcc -c main.c
generates
main.o
gcc -o main main.o -lm
generates
main
./main

**Figure 2.11** The role of the linker and loader.

command line on UNIX systems—for example, `./main`—the shell first creates a new process to run the program using the `fork()` system call. The shell then invokes the loader with the `exec()` system call, passing `exec()` the name of the executable file. The loader then loads the specified program into memory using the address space of the newly created process. (When a GUI interface is used, double-clicking on the icon associated with the executable file invokes the loader using a similar mechanism.)

The process described thus far assumes that all libraries are linked into the executable file and loaded into memory. In reality, most systems allow a program to dynamically link libraries as the program is loaded. Windows, for instance, supports dynamically linked libraries (**DLLs**). The benefit of this approach is that it avoids linking and loading libraries that may end up not being used into an executable file. Instead, the library is conditionally linked and is loaded if it is required during program run time. For example, in Figure 2.11, the math library is not linked into the executable file `main`. Rather, the linker inserts relocation information that allows it to be dynamically linked and loaded as the program is loaded. We shall see in Chapter 9 that it is possible for multiple processes to share dynamically linked libraries, resulting in a significant savings in memory use.

Object files and executable files typically have standard formats that include the compiled machine code and a symbol table containing metadata about functions and variables that are referenced in the program. For UNIX and Linux systems, this standard format is known as ELF (for **Executable and Linkable Format**). There are separate ELF formats for relocatable and

***ELF FORMAT***

Linux provides various commands to identify and evaluate ELF files. For example, the `file` command determines a file type. If `main.o` is an object file, and `main` is an executable file, the command

```
file main.o
```

will report that `main.o` is an ELF relocatable file, while the command

```
file main
```

will report that `main` is an ELF executable. ELF files are divided into a number of sections and can be evaluated using the `readelf` command.

executable files. One piece of information in the ELF file for executable files is the program's ***entry point,*** which contains the address of the first instruction to be executed when the program runs. Windows systems use the **Portable Executable** (PE) format, and macOS uses the **Mach-O** format.

## 2.6 Why Applications Are Operating-System Specific

Fundamentally, applications compiled on one operating system are not executable on other operating systems. If they were, the world would be a better place, and our choice of what operating system to use would depend on utility and features rather than which applications were available.

Based on our earlier discussion, we can now see part of the problem—each operating system provides a unique set of system calls. System calls are part of the set of services provided by operating systems for use by applications. Even if system calls were somehow uniform, other barriers would make it difficult for us to execute application programs on different operating systems. But if you have used multiple operating systems, you may have used some of the same applications on them. How is that possible?

An application can be made available to run on multiple operating systems in one of three ways:

1. The application can be written in an interpreted language (such as Python or Ruby) that has an interpreter available for multiple operating systems. The interpreter reads each line of the source program, executes equivalent instructions on the native instruction set, and calls native operating system calls. Performance suffers relative to that for native applications, and the interpreter provides only a subset of each operating system's features, possibly limiting the feature sets of the associated applications.
2. The application can be written in a language that includes a virtual machine containing the running application. The virtual machine is part of the language's full RTE. One example of this method is Java. Java has an RTE that includes a loader, byte-code verifier, and other components that load the Java application into the Java virtual machine. This RTE has been

**ported**, or developed, for many operating systems, from mainframes to smartphones, and in theory any Java app can run within the RTE wherever it is available. Systems of this kind have disadvantages similar to those of interpreters, discussed above.

3. The application developer can use a standard language or API in which the compiler generates binaries in a machine- and operating-system-specific language. The application must be ported to each operating system on which it will run. This porting can be quite time consuming and must be done for each new version of the application, with subsequent testing and debugging. Perhaps the best-known example is the POSIX API and its set of standards for maintaining source-code compatibility between different variants of UNIX-like operating systems.

In theory, these three approaches seemingly provide simple solutions for developing applications that can run across different operating systems. However, the general lack of application mobility has several causes, all of which still make developing cross-platform applications a challenging task. At the application level, the libraries provided with the operating system contain APIs to provide features like GUI interfaces, and an application designed to call one set of APIs (say, those available from iOS on the Apple iPhone) will not work on an operating system that does not provide those APIs (such as Android). Other challenges exist at lower levels in the system, including the following.

- Each operating system has a binary format for applications that dictates the layout of the header, instructions, and variables. Those components need to be at certain locations in specified structures within an executable file so the operating system can open the file and load the application for proper execution.
- CPUs have varying instruction sets, and only applications containing the appropriate instructions can execute correctly.
- Operating systems provide system calls that allow applications to request various activities, such as creating files and opening network connections. Those system calls vary among operating systems in many respects, including the specific operands and operand ordering used, how an application invokes the system calls, their numbering and number, their meanings, and their return of results.

There are some approaches that have helped address, though not completely solve, these architectural differences. For example, Linux—and almost every UNIX system—has adopted the ELF format for binary executable files. Although ELF provides a common standard across Linux and UNIX systems, the ELF format is not tied to any specific computer architecture, so it does not guarantee that an executable file will run across different hardware platforms.

APIs, as mentioned above, specify certain functions at the application level. At the architecture level, an **application binary interface** (ABI) is used to define how different components of binary code can interface for a given operating system on a given architecture. An ABI specifies low-level details, including address width, methods of passing parameters to system calls, the organization of the run-time stack, the binary format of system libraries, and the size of data types, just to name a few. Typically, an ABI is specified for a given architecture (for example, there is an ABI for the ARMv8 processor). Thus, an ABI is the architecture-level equivalent of an API. If a binary executable file has been compiled and linked according to a particular ABI, it should be able to run on different systems that support that ABI. However, because a particular ABI is defined for a certain operating system running on a given architecture, ABIs do little to provide cross-platform compatibility.

In sum, all of these differences mean that unless an interpreter, RTE, or binary executable file is written for and compiled on a specific operating system on a specific CPU type (such as Intel x86 or ARMv8), the application will fail to run. Imagine the amount of work that is required for a program such as the Firefox browser to run on Windows, macOS, various Linux releases, iOS, and Android, sometimes on various CPU architectures.

## 2.7 Operating-System Design and Implementation

In this section, we discuss problems we face in designing and implementing an operating system. There are, of course, no complete solutions to such problems, but there are approaches that have proved successful.

### 2.7.1 Design Goals

The first problem in designing a system is to define goals and specifications. At the highest level, the design of the system will be affected by the choice of hardware and the type of system: traditional desktop/laptop, mobile, distributed, or real time.

Beyond this highest design level, the requirements may be much harder to specify. The requirements can, however, be divided into two basic groups: **user goals** and **system goals**.

Users want certain obvious properties in a system. The system should be convenient to use, easy to learn and to use, reliable, safe, and fast. Of course, these specifications are not particularly useful in the system design, since there is no general agreement on how to achieve them.

A similar set of requirements can be defined by the developers who must design, create, maintain, and operate the system. The system should be easy to design, implement, and maintain; and it should be flexible, reliable, error free, and efficient. Again, these requirements are vague and may be interpreted in various ways.

There is, in short, no unique solution to the problem of defining the requirements for an operating system. The wide range of systems in existence shows that different requirements can result in a large variety of solutions for different environments. For example, the requirements for Wind River VxWorks, a real-time operating system for embedded systems, must have been substantially different from those for Windows Server, a large multiaccess operating system designed for enterprise applications.

Specifying and designing an operating system is a highly creative task. Although no textbook can tell you how to do it, general principles have been developed in the field of **software engineering**, and we turn now to a discussion of some of these principles.

### 2.7.2 Mechanisms and Policies

One important principle is the separation of **policy** from **mechanism**. Mechanisms determine *how* to do something; policies determine *what* will be done. For example, the timer construct (see Section 1.4.3) is a mechanism for ensuring CPU protection, but deciding how long the timer is to be set for a particular user is a policy decision.

The separation of policy and mechanism is important for flexibility. Policies are likely to change across places or over time. In the worst case, each change in policy would require a change in the underlying mechanism. A general mechanism flexible enough to work across a range of policies is preferable. A change in policy would then require redefinition of only certain parameters of the system. For instance, consider a mechanism for giving priority to certain types of programs over others. If the mechanism is properly separated from policy, it can be used either to support a policy decision that I/O-intensive programs should have priority over CPU-intensive ones or to support the opposite policy.

Microkernel-based operating systems (discussed in Section 2.8.3) take the separation of mechanism and policy to one extreme by implementing a basic set of primitive building blocks. These blocks are almost policy free, allowing more advanced mechanisms and policies to be added via user-created kernel modules or user programs themselves. In contrast, consider Windows, an enormously popular commercial operating system available for over three decades. Microsoft has closely encoded both mechanism and policy into the system to enforce a global look and feel across all devices that run the Windows operating system. All applications have similar interfaces, because the interface itself is built into the kernel and system libraries. Apple has adopted a similar strategy with its macOS and iOS operating systems.

We can make a similar comparison between commercial and open-source operating systems. For instance, contrast Windows, discussed above, with Linux, an open-source operating system that runs on a wide range of computing devices and has been available for over 25 years. The “standard” Linux kernel has a specific CPU scheduling algorithm (covered in Section 5.7.1), which is a mechanism that supports a certain policy. However, anyone is free to modify or replace the scheduler to support a different policy.

Policy decisions are important for all resource allocation. Whenever it is necessary to decide whether or not to allocate a resource, a policy decision must be made. Whenever the question is *how* rather than *what*, it is a mechanism that must be determined.

### 2.7.3 Implementation

Once an operating system is designed, it must be implemented. Because operating systems are collections of many programs, written by many people over a long period of time, it is difficult to make general statements about how they are implemented.

Early operating systems were written in assembly language. Now, most are written in higher-level languages such as C or C++, with small amounts of the system written in assembly language. In fact, more than one higher-level language is often used. The lowest levels of the kernel might be written in assembly language and C. Higher-level routines might be written in C and C++, and system libraries might be written in C++ or even higher-level languages. Android provides a nice example: its kernel is written mostly in C with some assembly language. Most Android system libraries are written in C or C++, and its application frameworks—which provide the developer interface to the system—are written mostly in Java. We cover Android's architecture in more detail in Section 2.8.5.2.

The advantages of using a higher-level language, or at least a systems-implementation language, for implementing operating systems are the same as those gained when the language is used for application programs: the code can be written faster, is more compact, and is easier to understand and debug. In addition, improvements in compiler technology will improve the generated code for the entire operating system by simple recompilation. Finally, an operating system is far easier to port to other hardware if it is written in a higher-level language. This is particularly important for operating systems that are intended to run on several different hardware systems, such as small embedded devices, Intel x86 systems, and ARM chips running on phones and tablets.

The only possible disadvantages of implementing an operating system in a higher-level language are reduced speed and increased storage requirements. This, however, is not a major issue in today's systems. Although an expert assembly-language programmer can produce efficient small routines, for large programs a modern compiler can perform complex analysis and apply sophisticated optimizations that produce excellent code. Modern processors have deep pipelining and multiple functional units that can handle the details of complex dependencies much more easily than can the human mind.

As is true in other systems, major performance improvements in operating systems are more likely to be the result of better data structures and algorithms than of excellent assembly-language code. In addition, although operating systems are large, only a small amount of the code is critical to high performance; the interrupt handlers, I/O manager, memory manager, and CPU scheduler are probably the most critical routines. After the system is written and is working correctly, bottlenecks can be identified and can be refactored to operate more efficiently.

## 2.8 Operating-System Structure

A system as large and complex as a modern operating system must be engineered carefully if it is to function properly and be modified easily. A common approach is to partition the task into small components, or modules, rather than have one single system. Each of these modules should be a well-defined portion of the system, with carefully defined interfaces and functions. You may use a similar approach when you structure your programs: rather than placing all of your code in the `main()` function, you instead separate logic into a number of functions, clearly articulate parameters and return values, and then call those functions from `main()`.

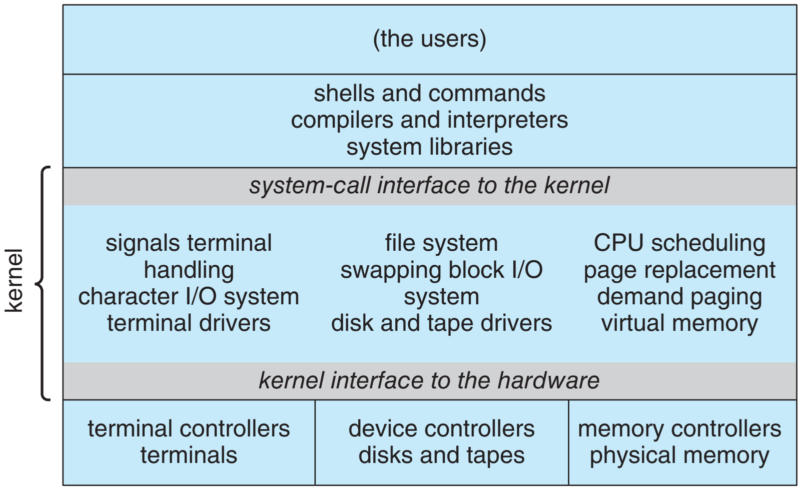

(the users)
shells and commands
compilers and interpreters
system libraries
system-call interface to the kernel
signals terminal handling
character I/O system
terminal drivers
file system
swapping block I/O system
disk and tape drivers
CPU scheduling
page replacement
demand paging
virtual memory
kernel interface to the hardware
terminal controllers
terminals
device controllers
disks and tapes
memory controllers
physical memory
kernel

**Figure 2.12** Traditional UNIX system structure.

We briefly discussed the common components of operating systems in Chapter 1. In this section, we discuss how these components are interconnected and melded into a kernel.

### 2.8.1 Monolithic Structure

The simplest structure for organizing an operating system is no structure at all. That is, place all of the functionality of the kernel into a single, static binary file that runs in a single address space. This approach—known as a **monolithic** structure—is a common technique for designing operating systems.

An example of such limited structuring is the original UNIX operating system, which consists of two separable parts: the kernel and the system programs. The kernel is further separated into a series of interfaces and device drivers, which have been added and expanded over the years as UNIX has evolved. We can view the traditional UNIX operating system as being layered to some extent, as shown in Figure 2.12. Everything below the system-call interface and above the physical hardware is the kernel. The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls. Taken in sum, that is an enormous amount of functionality to be combined into one single address space.

The Linux operating system is based on UNIX and is structured similarly, as shown in Figure 2.13. Applications typically use the `glibc` standard C library when communicating with the system call interface to the kernel. The Linux kernel is monolithic in that it runs entirely in kernel mode in a single address space, but as we shall see in Section 2.8.4, it does have a modular design that allows the kernel to be modified during run time.

Despite the apparent simplicity of monolithic kernels, they are difficult to implement and extend. Monolithic kernels do have a distinct performance advantage, however: there is very little overhead in the system-call interface, and communication within the kernel is fast. Therefore, despite the drawbacks

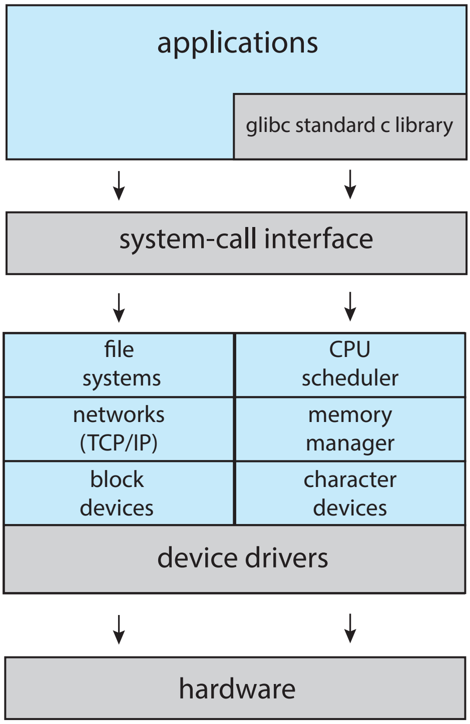

applications
glibc standard c library
system-call interface
file systems
CPU scheduler
networks (TCP/IP)
memory manager
block devices
character devices
device drivers
hardware

**Figure 2.13** Linux system structure.

of monolithic kernels, their speed and efficiency explains why we still see evidence of this structure in the UNIX, Linux, and Windows operating systems.

### 2.8.2 Layered Approach

The monolithic approach is often known as a **tightly coupled** system because changes to one part of the system can have wide-ranging effects on other parts. Alternatively, we could design a **loosely coupled** system. Such a system is divided into separate, smaller components that have specific and limited functionality. All these components together comprise the kernel. The advantage of this modular approach is that changes in one component affect only that component, and no others, allowing system implementers more freedom in creating and changing the inner workings of the system.

A system can be made modular in many ways. One method is the **layered approach**, in which the operating system is broken into a number of layers (levels). The bottom layer (layer 0) is the hardware; the highest (layer *N*) is the user interface. This layering structure is depicted in Figure 2.14.

An operating-system layer is an implementation of an abstract object made up of data and the operations that can manipulate those data. A typical operating-system layer—say, layer *M*—consists of data structures and a set of functions that can be invoked by higher-level layers. Layer *M*, in turn, can invoke operations on lower-level layers.

The main advantage of the layered approach is simplicity of construction and debugging. The layers are selected so that each uses functions (operations)

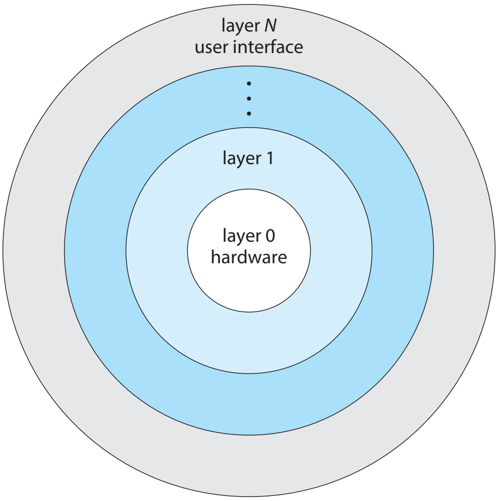


**Figure 2.14** A layered operating system.

and services of only lower-level layers. This approach simplifies debugging and system verification. The first layer can be debugged without any concern for the rest of the system, because, by definition, it uses only the basic hardware (which is assumed correct) to implement its functions. Once the first layer is debugged, its correct functioning can be assumed while the second layer is debugged, and so on. If an error is found during the debugging of a particular layer, the error must be on that layer, because the layers below it are already debugged. Thus, the design and implementation of the system are simplified.

Each layer is implemented only with operations provided by lower-level layers. A layer does not need to know how these operations are implemented; it needs to know only what these operations do. Hence, each layer hides the existence of certain data structures, operations, and hardware from higher-level layers.

Layered systems have been successfully used in computer networks (such as TCP/IP) and web applications. Nevertheless, relatively few operating systems use a pure layered approach. One reason involves the challenges of appropriately defining the functionality of each layer. In addition, the overall performance of such systems is poor due to the overhead of requiring a user program to traverse through multiple layers to obtain an operating-system service. ***Some*** layering is common in contemporary operating systems, however. Generally, these systems have fewer layers with more functionality, providing most of the advantages of modularized code while avoiding the problems of layer definition and interaction.

### 2.8.3 Microkernels

We have already seen that the original UNIX system had a monolithic structure. As UNIX expanded, the kernel became large and difficult to manage. In the mid-1980s, researchers at Carnegie Mellon University developed an operating system called **Mach** that modularized the kernel using the **microkernel** approach. This method structures the operating system by removing

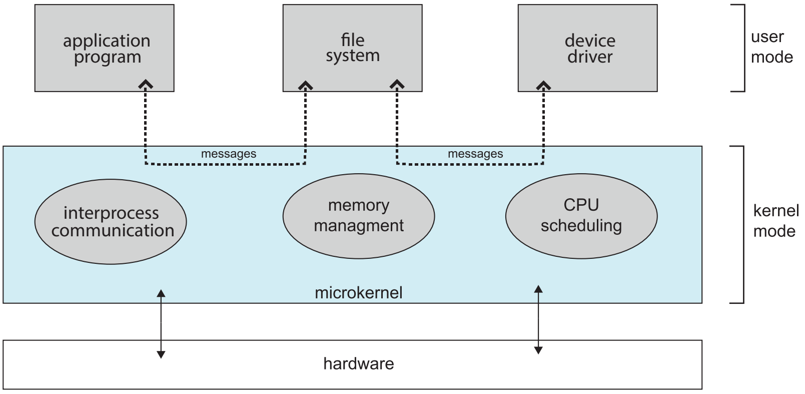


**Figure 2.15** Architecture of a typical microkernel.

all nonessential components from the kernel and implementing them as user-level programs that reside in separate address spaces. The result is a smaller kernel. There is little consensus regarding which services should remain in the kernel and which should be implemented in user space. Typically, however, microkernels provide minimal process and memory management, in addition to a communication facility. Figure 2.15 illustrates the architecture of a typical microkernel.

The main function of the microkernel is to provide communication between the client program and the various services that are also running in user space. Communication is provided through message passing, which was described in Section 2.3.3.5. For example, if the client program wishes to access a file, it must interact with the file server. The client program and service never interact directly. Rather, they communicate indirectly by exchanging messages with the microkernel.

One benefit of the microkernel approach is that it makes extending the operating system easier. All new services are added to user space and consequently do not require modification of the kernel. When the kernel does have to be modified, the changes tend to be fewer, because the microkernel is a smaller kernel. The resulting operating system is easier to port from one hardware design to another. The microkernel also provides more security and reliability, since most services are running as user—rather than kernel—processes. If a service fails, the rest of the operating system remains untouched.

Perhaps the best-known illustration of a microkernel operating system is ***Darwin***, the kernel component of the macOS and iOS operating systems. Darwin, in fact, consists of two kernels, one of which is the Mach microkernel. We will cover the macOS and iOS systems in further detail in Section 2.8.5.1.

Another example is QNX, a real-time operating system for embedded systems. The QNX Neutrino microkernel provides services for message passing and process scheduling. It also handles low-level network communication and hardware interrupts. All other services in QNX are provided by standard processes that run outside the kernel in user mode.

Unfortunately, the performance of microkernels can suffer due to increased system-function overhead. When two user-level services must communicate, messages must be copied between the services, which reside in separate address spaces. In addition, the operating system may have to switch from one process to the next to exchange the messages. The overhead involved in copying messages and switching between processes has been the largest impediment to the growth of microkernel-based operating systems. Consider the history of Windows NT: The first release had a layered microkernel organization. This version's performance was low compared with that of Windows 95. Windows NT 4.0 partially corrected the performance problem by moving layers from user space to kernel space and integrating them more closely. By the time Windows XP was designed, Windows architecture had become more monolithic than microkernel. Section 2.8.5.1 will describe how macOS addresses the performance issues of the Mach microkernel.

### 2.8.4 Modules

Perhaps the best current methodology for operating-system design involves using **loadable kernel modules** (**LKMs**). Here, the kernel has a set of core components and can link in additional services via modules, either at boot time or during run time. This type of design is common in modern implementations of UNIX, such as Linux, macOS, and Solaris, as well as Windows.

The idea of the design is for the kernel to provide core services, while other services are implemented dynamically, as the kernel is running. Linking services dynamically is preferable to adding new features directly to the kernel, which would require recompiling the kernel every time a change was made. Thus, for example, we might build CPU scheduling and memory management algorithms directly into the kernel and then add support for different file systems by way of loadable modules.

The overall result resembles a layered system in that each kernel section has defined, protected interfaces; but it is more flexible than a layered system, because any module can call any other module. The approach is also similar to the microkernel approach in that the primary module has only core functions and knowledge of how to load and communicate with other modules; but it is more efficient, because modules do not need to invoke message passing in order to communicate.

Linux uses loadable kernel modules, primarily for supporting device drivers and file systems. LKMs can be "inserted" into the kernel as the system is started (or ***booted***) or during run time, such as when a USB device is plugged into a running machine. If the Linux kernel does not have the necessary driver, it can be dynamically loaded. LKMs can be removed from the kernel during run time as well. For Linux, LKMs allow a dynamic and modular kernel, while maintaining the performance benefits of a monolithic system. We cover creating LKMs in Linux in several programming exercises at the end of this chapter.

### 2.8.5 Hybrid Systems

In practice, very few operating systems adopt a single, strictly defined structure. Instead, they combine different structures, resulting in hybrid systems that address performance, security, and usability issues. For example, Linux is monolithic, because having the operating system in a single address space provides very efficient performance. However, it also modular, so that new functionality can be dynamically added to the kernel. Windows is largely monolithic as well (again primarily for performance reasons), but it retains some behavior typical of microkernel systems, including providing support for separate subsystems (known as operating-system ***personalities***) that run as user-mode processes. Windows systems also provide support for dynamically loadable kernel modules. We provide case studies of Linux and Windows 10 in Chapter 20 and Chapter 21, respectively. In the remainder of this section, we explore the structure of three hybrid systems: the Apple macOS operating system and the two most prominent mobile operating systems—iOS and Android.

#### 2.8.5.1 macOS and iOS

Apple's macOS operating system is designed to run primarily on desktop and laptop computer systems, whereas iOS is a mobile operating system designed for the iPhone smartphone and iPad tablet computer. Architecturally, macOS and iOS have much in common, and so we present them together, highlighting what they share as well as how they differ from each other. The general architecture of these two systems is shown in Figure 2.16. Highlights of the various layers include the following:

- **User experience layer**. This layer defines the software interface that allows users to interact with the computing devices. macOS uses the ***Aqua*** user interface, which is designed for a mouse or trackpad, whereas iOS uses the ***Springboard*** user interface, which is designed for touch devices.
- **Application frameworks layer**. This layer includes the ***Cocoa*** and ***Cocoa Touch*** frameworks, which provide an API for the Objective-C and Swift programming languages. The primary difference between Cocoa and Cocoa Touch is that the former is used for developing macOS applications, and the latter by iOS to provide support for hardware features unique to mobile devices, such as touch screens.
- **Core frameworks**. This layer defines frameworks that support graphics and media including, Quicktime and OpenGL.

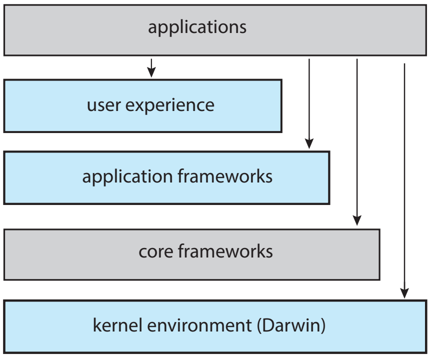


**Figure 2.16** Architecture of Apple's macOS and iOS operating systems.

- **Kernel environment**. This environment, also known as **Darwin**, includes the Mach microkernel and the BSD UNIX kernel. We will elaborate on Darwin shortly.

As shown in Figure 2.16, applications can be designed to take advantage of user-experience features or to bypass them and interact directly with either the application framework or the core framework. Additionally, an application can forego frameworks entirely and communicate directly with the kernel environment. (An example of this latter situation is a C program written with no user interface that makes POSIX system calls.)

Some significant distinctions between macOS and iOS include the following:

- Because macOS is intended for desktop and laptop computer systems, it is compiled to run on Intel architectures. iOS is designed for mobile devices and thus is compiled for ARM-based architectures. Similarly, the iOS kernel has been modified somewhat to address specific features and needs of mobile systems, such as power management and aggressive memory management. Additionally, iOS has more stringent security settings than macOS.
- The iOS operating system is generally much more restricted to developers than macOS and may even be closed to developers. For example, iOS restricts access to POSIX and BSD APIs on iOS, whereas they are openly available to developers on macOS.

We now focus on Darwin, which uses a hybrid structure. Darwin is a layered system that consists primarily of the Mach microkernel and the BSD UNIX kernel. Darwin's structure is shown in Figure 2.17.

Whereas most operating systems provide a single system-call interface to the kernel—such as through the standard C library on UNIX and Linux systems—Darwin provides *two* system-call interfaces: Mach system calls (known as

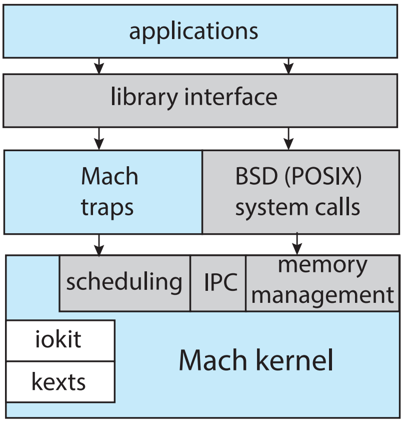


**Figure 2.17** The structure of Darwin.

**traps**) and BSD system calls (which provide POSIX functionality). The interface to these system calls is a rich set of libraries that includes not only the standard C library but also libraries that provide networking, security, and progamming language support (to name just a few).

Beneath the system-call interface, Mach provides fundamental operating-system services, including memory management, CPU scheduling, and interprocess communication (IPC) facilities such as message passing and remote procedure calls (RPCs). Much of the functionality provided by Mach is available through **kernel abstractions**, which include tasks (a Mach process), threads, memory objects, and ports (used for IPC). As an example, an application may create a new process using the BSD POSIX `fork()` system call. Mach will, in turn, use a task kernel abstraction to represent the process in the kernel.

In addition to Mach and BSD, the kernel environment provides an I/O kit for development of device drivers and dynamically loadable modules (which macOS refers to as **kernel extensions**, or **kexts**).

In Section 2.8.3, we described how the overhead of message passing between different services running in user space compromises the performance of microkernels. To address such performance problems, Darwin combines Mach, BSD, the I/O kit, and any kernel extensions into a single address space. Thus, Mach is not a pure microkernel in the sense that various subsystems run in user space. Message passing within Mach still does occur, but no copying is necessary, as the services have access to the same address space.

Apple has released the Darwin operating system as open source. As a result, various projects have added extra functionality to Darwin, such as the X-11 windowing system and support for additional file systems. Unlike Darwin, however, the Cocoa interface, as well as other proprietary Apple frameworks available for developing macOS applications, are closed.

#### 2.8.5.2 Android

The Android operating system was designed by the Open Handset Alliance (led primarily by Google) and was developed for Android smartphones and tablet computers. Whereas iOS is designed to run on Apple mobile devices and is close-sourced, Android runs on a variety of mobile platforms and is open-sourced, partly explaining its rapid rise in popularity. The structure of Android appears in Figure 2.18.

Android is similar to iOS in that it is a layered stack of software that provides a rich set of frameworks supporting graphics, audio, and hardware features. These features, in turn, provide a platform for developing mobile applications that run on a multitude of Android-enabled devices.

Software designers for Android devices develop applications in the Java language, but they do not generally use the standard Java API. Google has designed a separate Android API for Java development. Java applications are compiled into a form that can execute on the Android RunTime ART, a virtual machine designed for Android and optimized for mobile devices with limited memory and CPU processing capabilities. Java programs are first compiled to a Java bytecode `.class` file and then translated into an executable `.dex` file. Whereas many Java virtual machines perform just-in-time (JIT) compilation to improve application efficiency, ART performs **ahead-of-time** (**AOT**) compila-

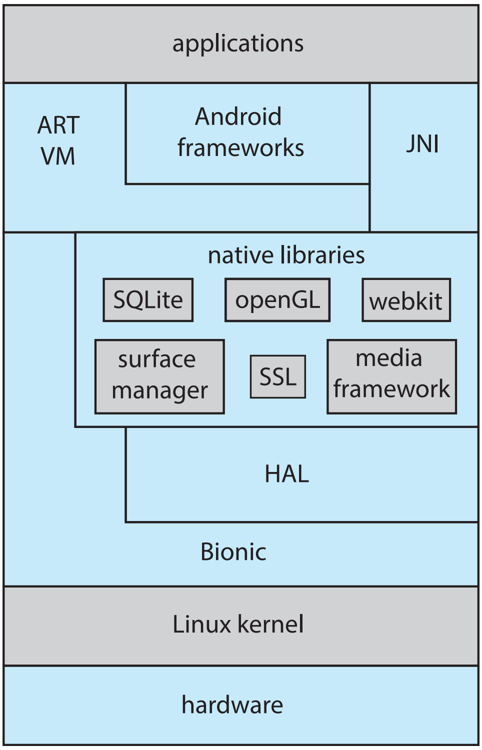


**Figure 2.18** Architecture of Google's Android.

tion. Here, `.dex` files are compiled into native machine code when they are installed on a device, from which they can execute on the ART. AOT compilation allows more efficient application execution as well as reduced power consumption, features that are crucial for mobile systems.

Android developers can also write Java programs that use the Java native interface—or JNI—which allows developers to bypass the virtual machine and instead write Java programs that can access specific hardware features. Programs written using JNI are generally not portable from one hardware device to another.

The set of native libraries available for Android applications includes frameworks for developing web browsers (webkit), database support (SQLite), and network support, such as secure sockets (SSLs).

Because Android can run on an almost unlimited number of hardware devices, Google has chosen to abstract the physical hardware through the hardware abstraction layer, or HAL. By abstracting all hardware, such as the camera, GPS chip, and other sensors, the HAL provides applications with a consistent view independent of specific hardware. This feature, of course, allows developers to write programs that are portable across different hardware platforms.

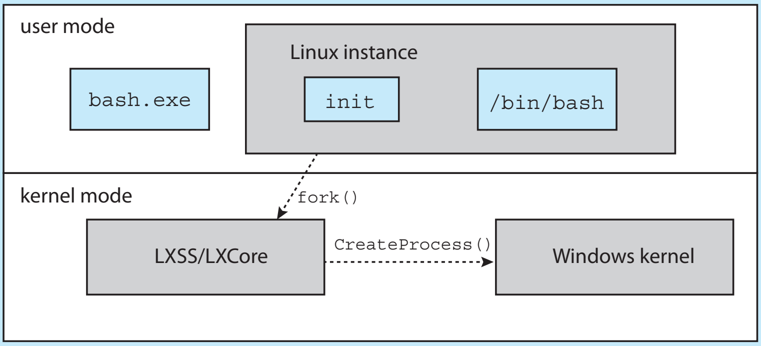
The standard C library used by Linux systems is the GNU C library (`glibc`). Google instead developed the **Bionic** standard C library for Android. Not only does Bionic have a smaller memory footprint than `glibc`, but it also has been designed for the slower CPUs that characterize mobile devices. (In addition, Bionic allows Google to bypass GPL licensing of `glibc`.)

At the bottom of Android's software stack is the Linux kernel. Google has modified the Linux kernel used in Android in a variety of areas to support the special needs of mobile systems, such as power management. It has also made changes in memory management and allocation and has added a new form of IPC known as ***Binder*** (which we will cover in Section 3.8.2.1).

### *WINDOWS SUBSYSTEM FOR LINUX*

Windows uses a hybrid architecture that provides subsystems to emulate different operating-system environments. These user-mode subsystems communicate with the Windows kernel to provide actual services. Windows 10 adds a Windows subsystem for Linux (**WSL**), which allows native Linux applications (specified as ELF binaries) to run on Windows 10. The typical operation is for a user to start the Windows application `bash.exe`, which presents the user with a `bash` shell running Linux. Internally, the WSL creates a **Linux instance** consisting of the `init` process, which in turn creates the `bash` shell running the native Linux application `/bin/bash`. Each of these processes runs in a Windows **Pico** process. This special process loads the native Linux binary into the process's own address space, thus providing an environment in which a Linux application can execute.

Pico processes communicate with the kernel services LXCore and LXSS to translate Linux system calls, if possible using native Windows system calls. When the Linux application makes a system call that has no Windows equivalent, the LXSS service must provide the equivalent functionality. When there is a one-to-one relationship between the Linux and Windows system calls, LXSS forwards the Linux system call directly to the equivalent call in the Windows kernel. In some situations, Linux and Windows have system calls that are similar but not identical. When this occurs, LXSS will provide some of the functionality and will invoke the similar Windows system call to provide the remainder of the functionality. The Linux `fork()` provides an illustration of this: The Windows `CreateProcess()` system call is similar to `fork()` but does not provide exactly the same functionality. When `fork()` is invoked in WSL, the LXSS service does some of the initial work of `fork()` and then calls `CreateProcess()` to do the remainder of the work. The figure below illustrates the basic behavior of WSL.



## 2.9 Building and Booting an Operating System

It is possible to design, code, and implement an operating system specifically for one specific machine configuration. More commonly, however, operating systems are designed to run on any of a class of machines with a variety of peripheral configurations.

### 2.9.1 Operating-System Generation

Most commonly, a computer system, when purchased, has an operating system already installed. For example, you may purchase a new laptop with Windows or macOS preinstalled. But suppose you wish to replace the preinstalled operating system or add additional operating systems. Or suppose you purchase a computer without an operating system. In these latter situations, you have a few options for placing the appropriate operating system on the computer and configuring it for use.

If you are generating (or building) an operating system from scratch, you must follow these steps:

1. Write the operating system source code (or obtain previously written source code).
2. Configure the operating system for the system on which it will run.
3. Compile the operating system.
4. Install the operating system.
5. Boot the computer and its new operating system.

Configuring the system involves specifying which features will be included, and this varies by operating system. Typically, parameters describing how the system is configured is stored in a configuration file of some type, and once this file is created, it can be used in several ways.

At one extreme, a system administrator can use it to modify a copy of the operating-system source code. Then the operating system is completely compiled (known as a **system build**). Data declarations, initializations, and constants, along with compilation, produce an output-object version of the operating system that is tailored to the system described in the configuration file.

At a slightly less tailored level, the system description can lead to the selection of precompiled object modules from an existing library. These modules are linked together to form the generated operating system. This process allows the library to contain the device drivers for all supported I/O devices, but only those needed are selected and linked into the operating system. Because the system is not recompiled, system generation is faster, but the resulting system may be overly general and may not support different hardware configurations.

At the other extreme, it is possible to construct a system that is completely modular. Here, selection occurs at execution time rather than at compile or link time. System generation involves simply setting the parameters that describe the system configuration.

The major differences among these approaches are the size and generality of the generated system and the ease of modifying it as the hardware configuration changes. For embedded systems, it is not uncommon to adopt the first approach and create an operating system for a specific, static hardware configuration. However, most modern operating systems that support desktop and laptop computers as well as mobile devices have adopted the second approach. That is, the operating system is still generated for a specific hardware configuration, but the use of techniques such as loadable kernel modules provides modular support for dynamic changes to the system.

We now illusrate how to build a Linux system from scratch, where it is typically necessary to perform the following steps:

1. Download the Linux source code from http://www.kernel.org.
2. Configure the kernel using the "`make menuconfig`" command. This step generates the `.config` configuration file.
3. Compile the main kernel using the "`make`" command. The `make` command compiles the kernel based on the configuration parameters identified in the `.config` file, producing the file `vmlinuz`, which is the kernel image.
4. Compile the kernel modules using the "`make modules`" command. Just as with compiling the kernel, module compilation depends on the configuration parameters specified in the `.config` file.
5. Use the command "`make modules_install`" to install the kernel modules into `vmlinuz`.
6. Install the new kernel on the system by entering the "`make install`" command.

When the system reboots, it will begin running this new operating system.

Alternatively, it is possible to modify an existing system by installing a Linux virtual machine. This will allow the host operating system (such as Windows or macOS) to run Linux. (We introduced virtualization in Section 1.7 and cover the topic more fully in Chapter 18.)

There are a few options for installing Linux as a virtual machine. One alternative is to build a virtual machine from scratch. This option is similar to building a Linux system from scratch; however, the operating system does not need to be compiled. Another approach is to use a Linux virtual machine appliance, which is an operating system that has already been built and configured. This option simply requires downloading the appliance and installing it using virtualization software such as VirtualBox or VMware. For example, to build the operating system used in the virtual machine provided with this text, the authors did the following:

1. Downloaded the Ubuntu ISO image from https://www.ubuntu.com/
2. Instructed the virtual machine software VirtualBox to use the ISO as the bootable medium and booted the virtual machine
3. Answered the installation questions and then installed and booted the operating system as a virtual machine

### 2.9.2 System Boot

After an operating system is generated, it must be made available for use by the hardware. But how does the hardware know where the kernel is or how to load that kernel? The process of starting a computer by loading the kernel is known as **booting** the system. On most systems, the boot process proceeds as follows:

1. A small piece of code known as the **bootstrap program** or **boot loader** locates the kernel.
2. The kernel is loaded into memory and started.
3. The kernel initializes hardware.
4. The root file system is mounted.

In this section, we briefly describe the boot process in more detail.

Some computer systems use a multistage boot process: When the computer is first powered on, a small boot loader located in nonvolatile firmware known as **BIOS** is run. This initial boot loader usually does nothing more than load a second boot loader, which is located at a fixed disk location called the **boot block**. The program stored in the boot block may be sophisticated enough to load the entire operating system into memory and begin its execution. More typically, it is simple code (as it must fit in a single disk block) and knows only the address on disk and the length of the remainder of the bootstrap program.

Many recent computer systems have replaced the BIOS-based boot process with **UEFI** (Unified Extensible Firmware Interface). UEFI has several advantages over BIOS, including better support for 64-bit systems and larger disks. Perhaps the greatest advantage is that UEFI is a single, complete boot manager and therefore is faster than the multistage BIOS boot process.

Whether booting from BIOS or UEFI, the bootstrap program can perform a variety of tasks. In addition to loading the file containing the kernel program into memory, it also runs diagnostics to determine the state of the machine —for example, inspecting memory and the CPUand discovering devices. If the diagnostics pass, the program can continue with the booting steps. The bootstrap can also initialize all aspects of the system, from CPU registers to device controllers and the contents of main memory. Sooner or later, it starts the operating system and mounts the root file system. It is only at this point is the system said to be **running**.

**GRUB** is an open-source bootstrap program for Linux and UNIX systems. Boot parameters for the system are set in a GRUB configuration file, which is loaded at startup. GRUB is flexible and allows changes to be made at boot time, including modifying kernel parameters and even selecting among different kernels that can be booted. As an example, the following are kernel parameters from the special Linux file `/proc/cmdline`, which is used at boot time:

```
BOOT_IMAGE=/boot/vmlinuz-4.4.0-59-generic
root=UUID=5f2e2232-4e47-4fe8-ae94-45ea749a5c92
```

`BOOT_IMAGE` is the name of the kernel image to be loaded into memory, and `root` specifies a unique identifier of the root file system.

To save space as well as decrease boot time, the Linux kernel image is a compressed file that is extracted after it is loaded into memory. During the boot process, the boot loader typically creates a temporary RAM file system, known as `initramfs`. This file system contains necessary drivers and kernel modules that must be installed to support the ***real*** root file system (which is not in main memory). Once the kernel has started and the necessary drivers are installed, the kernel switches the root file system from the temporary RAM location to the appropriate root file system location. Finally, Linux creates the `systemd` process, the initial process in the system, and then starts other services (for example, a web server and/or database). Ultimately, the system will present the user with a login prompt. In Section 11.5.2, we describe the boot process for Windows.

It is worthwhile to note that the booting mechanism is not independent from the boot loader. Therefore, there are specific versions of the GRUB boot loader for BIOS and UEFI, and the firmware must know as well which specific bootloader is to be used.

The boot process for mobile systems is slightly different from that for traditional PCs. For example, although its kernel is Linux-based, Android does not use GRUB and instead leaves it up to vendors to provide boot loaders. The most common Android boot loader is LK (for "little kernel"). Android systems use the same compressed kernel image as Linux, as well as an initial RAM file system. However, whereas Linux discards the `initramfs` once all necessary drivers have been loaded, Android maintains `initramfs` as the root file system for the device. Once the kernel has been loaded and the root file system mounted, Android starts the `init` process and creates a number of services before displaying the home screen.

Finally, boot loaders for most operating systems—including Windows, Linux, and macOS, as well as both iOS and Android—provide booting into **recovery mode** or **single-user mode** for diagnosing hardware issues, fixing corrupt file systems, and even reinstalling the operating system. In addition to hardware failures, computer systems can suffer from software errors and poor operating-system performance, which we consider in the following section.

## 2.10 Operating-System Debugging

We have mentioned debugging from time to time in this chapter. Here, we take a closer look. Broadly, **debugging** is the activity of finding and fixing errors in a system, both in hardware and in software. Performance problems are considered bugs, so debugging can also include **performance tuning**, which seeks to improve performance by removing processing **bottlenecks**. In this section, we explore debugging process and kernel errors and performance problems. Hardware debugging is outside the scope of this text.

### 2.10.1 Failure Analysis

If a process fails, most operating systems write the error information to a **log file** to alert system administrators or users that the problem occurred. The operating system can also take a **core dump**—a capture of the memory of the process—and store it in a file for later analysis. (Memory was referred to as the "core" in the early days of computing.) Running programs and core dumps can be probed by a debugger, which allows a programmer to explore the code and memory of a process at the time of failure.

Debugging user-level process code is a challenge. Operating-system kernel debugging is even more complex because of the size and complexity of the kernel, its control of the hardware, and the lack of user-level debugging tools. A failure in the kernel is called a **crash**. When a crash occurs, error information is saved to a log file, and the memory state is saved to a **crash dump**.

Operating-system debugging and process debugging frequently use different tools and techniques due to the very different nature of these two tasks. Consider that a kernel failure in the file-system code would make it risky for the kernel to try to save its state to a file on the file system before rebooting. A common technique is to save the kernel's memory state to a section of disk set aside for this purpose that contains no file system. If the kernel detects an unrecoverable error, it writes the entire contents of memory, or at least the kernel-owned parts of the system memory, to the disk area. When the system reboots, a process runs to gather the data from that area and write it to a crash dump file within a file system for analysis. Obviously, such strategies would be unnecessary for debugging ordinary user-level processes.

### 2.10.2 Performance Monitoring and Tuning

We mentioned earlier that performance tuning seeks to improve performance by removing processing bottlenecks. To identify bottlenecks, we must be able to monitor system performance. Thus, the operating system must have some means of computing and displaying measures of system behavior. Tools may be characterized as providing either ***per-process*** or ***system-wide*** observations. To make these observations, tools may use one of two approaches—***counters*** or ***tracing***. We explore each of these in the following sections.

#### 2.10.2.1 Counters

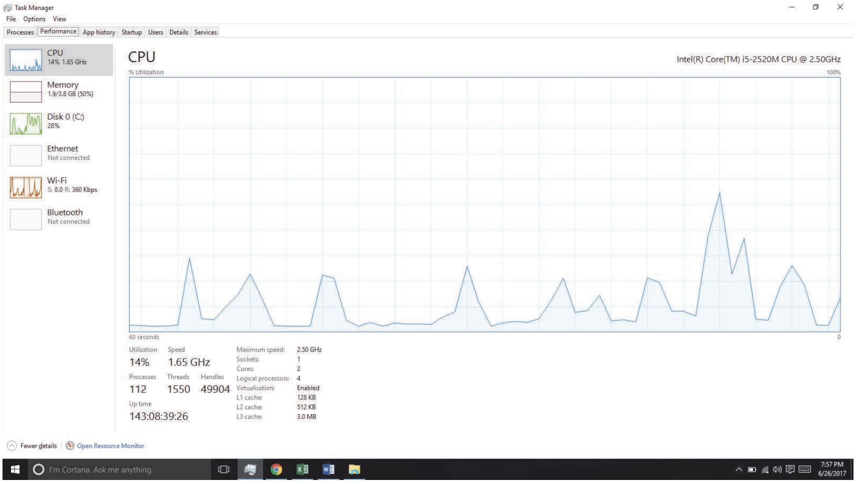
Operating systems keep track of system activity through a series of counters, such as the number of system calls made or the number of operations performed to a network device or disk. The following are examples of Linux tools that use counters:

**Per-Process**

- `ps`—reports information for a single process or selection of processes
- `top`—reports real-time statistics for current processes

**System-Wide**

- `vmstat`—reports memory-usage statistics
- `netstat`—reports statistics for network interfaces
- `iostat`—reports I/O usage for disks


Task Manager
File Options View
Processes Performance App history Startup Users Details Services
CPU
14% 1.65 GHz
Memory
1.9/3.8 GB (50%)
Disk 0 (C:)
28%
Ethernet
Not connected
Wi-Fi
S: 8.0 R: 360 Kbps
Bluetooth
Not connected
CPU
Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz
% Utilization
100%
60 seconds
0
Fewer details
Open Resource Monitor
I'm Cortana. Ask me anything.
7:57 PM
6/26/2017

Utilization 14%
Speed 1.65 GHz
Processes 112
Threads 1550
Handles 49904
Up time 143:08:39:26
Maximum speed: 2.50 GHz
Sockets: 1
Cores: 2
Logical processors: 4
Virtualization: Enabled
L1 cache: 128 KB
L2 cache: 512 KB
L3 cache: 3.0 MB

**Figure 2.19** The Windows 10 task manager.

Most of the counter-based tools on Linux systems read statistics from the `/proc` file system. `/proc` is a "pseudo" file system that exists only in kernel memory and is used primarily for querying various per-process as well as kernel statistics. The `/proc` file system is organized as a directory hierarchy, with the process (a unique integer value assigned to each process) appearing as a subdirectory below `/proc`. For example, the directory entry `/proc/2155` would contain per-process statistics for the process with an ID of 2155. There are `/proc` entries for various kernel statistics as well. In both this chapter and Chapter 3, we provide programming projects where you will create and access the `/proc` file system.

Windows systems provide the **Windows Task Manager**, a tool that includes information for current applications as well as processes, CPU and memory usage, and networking statistics. A screen shot of the task manager in Windows 10 appears in Figure 2.19.

### 2.10.3 Tracing

Whereas counter-based tools simply inquire on the current value of certain statistics that are maintained by the kernel, tracing tools collect data for a specific event—such as the steps involved in a system-call invocation.

The following are examples of Linux tools that trace events:

#### Per-Process

- `strace`—traces system calls invoked by a process
- `gdb`—a source-level debugger

#### System-Wide

- `perf`—a collection of Linux performance tools
- `tcpdump`—collects network packets

### *Kernighan's Law*

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

Making operating systems easier to understand, debug, and tune as they run is an active area of research and practice. A new generation of kernel-enabled performance analysis tools has made significant improvements in how this goal can be achieved. Next, we discuss BCC, a toolkit for dynamic kernel tracing in Linux.

### 2.10.4 BCC

Debugging the interactions between user-level and kernel code is nearly impossible without a toolset that understands both sets of code and can instrument their interactions. For that toolset to be truly useful, it must be able to debug any area of a system, including areas that were not written with debugging in mind, and do so without affecting system reliability. This toolset must also have a minimal performance impact—ideally it should have no impact when not in use and a proportional impact during use. The BCC toolkit meets these requirements and provides a dynamic, secure, low-impact debugging environment.

**BCC** (BPF Compiler Collection) is a rich toolkit that provides tracing features for Linux systems. BCC is a front-end interface to the eBPF (extended Berkeley Packet Filter) tool. The BPF technology was developed in the early 1990s for filtering traffic across a computer network. The "extended" BPF (eBPF) added various features to BPF. eBPF programs are written in a subset of C and are compiled into eBPF instructions, which can be dynamically inserted into a running Linux system. The eBPF instructions can be used to capture specific events (such as a certain system call being invoked) or to monitor system performance (such as the time required to perform disk I/O). To ensure that eBPF instructions are well behaved, they are passed through a **verifier** before being inserted into the running Linux kernel. The verifier checks to make sure that the instructions do not affect system performance or security.

Although eBPF provides a rich set of features for tracing within the Linux kernel, it traditionally has been very difficult to develop programs using its C interface. BCC was developed to make it easier to write tools using eBPF by providing a front-end interface in Python. A BCC tool is written in Python and it embeds C code that interfaces with the eBPF instrumentation, which in turn interfaces with the kernel. The BCC tool also compiles the C program into eBPF instructions and inserts it into the kernel using either probes or tracepoints, two techniques that allow tracing events in the Linux kernel.

The specifics of writing custom BCC tools are beyond the scope of this text, but the BCC package (which is installed on the Linux virtual machine we provide) provides a number of existing tools that monitor several areas of activity in a running Linux kernel. As an example, the BCC `disksnoop` tool traces disk I/O activity. Entering the command

```
./disksnoop.py
```

generates the following example output:

```
TIME(s)            T       BYTES       LAT(ms)
1946.29186700      R       8              0.27
1946.33965000      R       8              0.26
1948.34585000      W       8192           0.96
1950.43251000      R       4096           0.56
1951.74121000      R       4096           0.35
```

This output tells us the timestamp when the I/O operation occurred, whether the I/O was a `R`ead or `W`rite operation, and how many bytes were involved in the I/O. The final column reflects the duration (expressed as latency or `LAT`) in milliseconds of the I/O.

Many of the tools provided by BCC can be used for specific applications, such as MySQL databases, as well as Java and Python programs. Probes can also be placed to monitor the activity of a specific process. For example, the command

```
./opensnoop -p 1225
```

will trace `open()` system calls performed only by the process with an identifier of 1225.

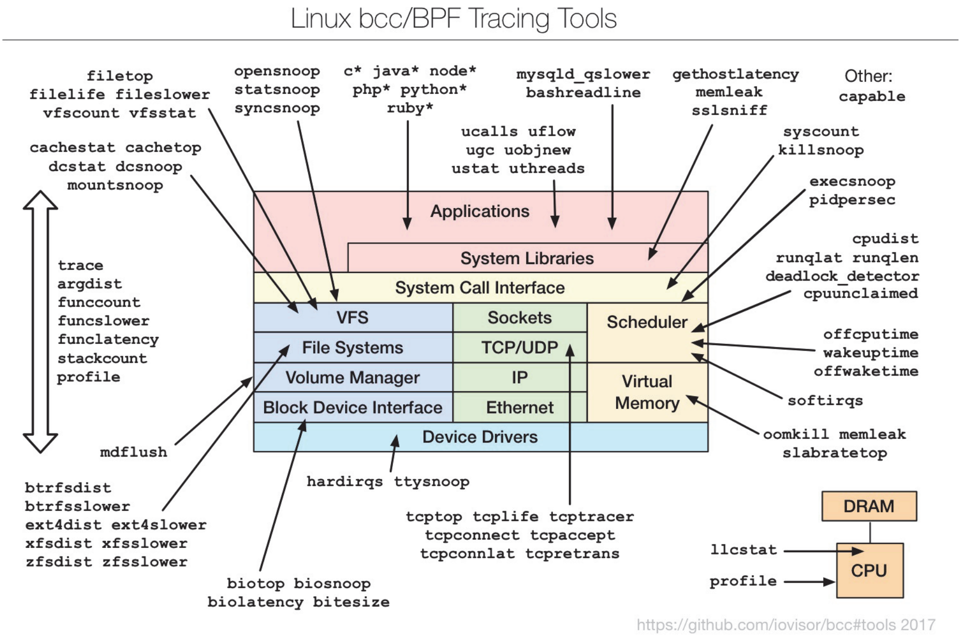

Linux bcc/BPF Tracing Tools
filetop
filelife fileslower
vfscount vfsstat
cachestat cachetop
dcstat dcsnoop
mountsnoop
opensnoop
statsnoop
syncsnoop
c* java* node*
php* python*
ruby*
ucalls uflow
ugc uobjnew
ustat uthreads
mysqld_qslower
bashreadline
gethostlatency
memleak
sslsniff
Other:
capable
syscount
killsnoop
execsnoop
pidpersec
cpudist
runqlat runqlen
deadlock_detector
cpuunclaimed
Applications
System Libraries
System Call Interface
VFS
Sockets
Scheduler
File Systems
TCP/UDP
Volume Manager
IP
Virtual
Memory
Block Device Interface
Ethernet
Device Drivers
trace
argdist
funccount
funcslower
funclatency
stackcount
profile
offcputime
wakeuptime
offwaketime
softirqs
oomkill memleak
slabratetop
mdflush
btrfsdist
btrfsslower
ext4dist ext4slower
xfsdist xfsslower
zfsdist zfsslower
hardirqs ttysnoop
tcptop tcplife tcptracer
tcpconnect tcpaccept
tcpconnlat tcpretrans
biotop biosnoop
biolatency bitesize
DRAM
llcstat
profile
CPU
https://github.com/iovisor/bcc#tools 2017

**Figure 2.20** The BCC and eBPF tracing tools.

What makes BCC especially powerful is that its tools can be used on live production systems that are running critical applications without causing harm to the system. This is particularly useful for system administrators who must monitor system performance to identify possible bottlenecks or security exploits. Figure 2.20 illustrates the wide range of tools currently provided by BCC and eBPF and their ability to trace essentially any area of the Linux operating system. BCC is a rapidly changing technology with new features constantly being added.

## 2.11 Summary

- An operating system provides an environment for the execution of programs by providing services to users and programs.
- The three primary approaches for interacting with an operating system are (1) command interpreters, (2) graphical user interfaces, and (3) touchscreen interfaces.
- System calls provide an interface to the services made available by an operating system. Programmers use a system call's application programming interface (API) for accessing system-call services.
- System calls can be divided into six major categories: (1) process control, (2) file management, (3) device management, (4) information maintenance, (5) communications, and (6) protection.
- The standard C library provides the system-call interface for UNIX and Linux systems.
- Operating systems also include a collection of system programs that provide utilities to users.
- A linker combines several relocatable object modules into a single binary executable file. A loader loads the executable file into memory, where it becomes eligible to run on an available CPU.
- There are several reasons why applications are operating-system specific. These include different binary formats for program executables, different instruction sets for different CPUs, and system calls that vary from one operating system to another.
- An operating system is designed with specific goals in mind. These goals ultimately determine the operating system's policies. An operating system implements these policies through specific mechanisms.
- A monolithic operating system has no structure; all functionality is provided in a single, static binary file that runs in a single address space. Although such systems are difficult to modify, their primary benefit is efficiency.
- A layered operating system is divided into a number of discrete layers, where the bottom layer is the hardware interface and the highest layer is the user interface. Although layered software systems have had some suc-

cess, this approach is generally not ideal for designing operating systems due to performance problems.

- The microkernel approach for designing operating systems uses a minimal kernel; most services run as user-level applications. Communication takes place via message passing.
- A modular approach for designing operating systems provides operating-system services through modules that can be loaded and removed during run time. Many contemporary operating systems are constructed as hybrid systems using a combination of a monolithic kernel and modules.
- A boot loader loads an operating system into memory, performs initialization, and begins system execution.
- The performance of an operating system can be monitored using either counters or tracing. Counters are a collection of system-wide or per-process statistics, while tracing follows the execution of a program through the operating system.

## Practice Exercises

**2.1** What is the purpose of system calls?

**2.2** What is the purpose of the command interpreter? Why is it usually separate from the kernel?

**2.3** What system calls have to be executed by a command interpreter or shell in order to start a new process on a UNIX system?

**2.4** What is the purpose of system programs?

**2.5** What is the main advantage of the layered approach to system design? What are the disadvantages of the layered approach?

**2.6** List five services provided by an operating system, and explain how each creates convenience for users. In which cases would it be impossible for user-level programs to provide these services? Explain your answer.

**2.7** Why do some systems store the operating system in firmware, while others store it on disk?

**2.8** How could a system be designed to allow a choice of operating systems from which to boot? What would the bootstrap program need to do?

## Further Reading

[Bryant and O'Hallaron (2015)] provide an overview of computer systems, including the role of the linker and loader. [Atlidakis et al. (2016)] discuss POSIX system calls and how they relate to modern operating systems. [Levin (2013)] covers the internals of both macOS and iOS, and [Levin (2015)] describes details of the Android system. Windows 10 internals are covered in [Russinovich et al. (2017)]. BSD UNIX is described in [McKusick et al. (2015)]. [Love (2010)] and

[Mauerer (2008)] thoroughly discuss the Linux kernel. Solaris is fully described in [McDougall and Mauro (2007)].

Linux source code is available at http://www.kernel.org. The Ubuntu ISO image is available from https://www.ubuntu.com/.

Comprehensive coverage of Linux kernel modules can be found at http://www.tldp.org/LDP/lkmpg/2.6/lkmpg.pdf. [Ward (2015)] and http://www.ibm.com/developerworks/linux/library/l-linuxboot/ describe the Linux boot process using GRUB. Performance tuning—with a focus on Linux and Solaris systems—is covered in [Gregg (2014)]. Details for the BCC toolkit can be found at https://github.com/iovisor/bcc/#tools.

## Bibliography

**[Atlidakis et al. (2016)]** V. Atlidakis, J. Andrus, R. Geambasu, D. Mitropoulos, and J. Nieh, “POSIX Abstractions in Modern Operating Systems: The Old, the New, and the Missing” (2016), pages 19:1–19:17.

**[Bryant and O’Hallaron (2015)]** R. Bryant and D. O’Hallaron, *Computer Systems: A Programmer’s Perspective*, Third Edition (2015).

**[Gregg (2014)]** B. Gregg, *Systems Performance–Enterprise and the Cloud*, Pearson (2014).

**[Levin (2013)]** J. Levin, *Mac OS X and iOS Internals to the Apple’s Core*, Wiley (2013).

**[Levin (2015)]** J. Levin, *Android Internals–A Confectioner’s Cookbook. Volume I* (2015).

**[Love (2010)]** R. Love, *Linux Kernel Development*, Third Edition, Developer’s Library (2010).

**[Mauerer (2008)]** W. Mauerer, *Professional Linux Kernel Architecture*, John Wiley and Sons (2008).

**[McDougall and Mauro (2007)]** R. McDougall and J. Mauro, *Solaris Internals*, Second Edition, Prentice Hall (2007).

**[McKusick et al. (2015)]** M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson, *The Design and Implementation of the FreeBSD UNIX Operating System–Second Edition*, Pearson (2015).

**[Russinovich et al. (2017)]** M. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals–Part 1*, Seventh Edition, Microsoft Press (2017).

**[Ward (2015)]** B. Ward, *How LINUX Works–What Every Superuser Should Know*, Second Edition, No Starch Press (2015).

## Chapter 2 Exercises

**2.9** The services and functions provided by an operating system can be divided into two main categories. Briefly describe the two categories, and discuss how they differ.

**2.10** Describe three general methods for passing parameters to the operating system.

**2.11** Describe how you could obtain a statistical profile of the amount of time a program spends executing different sections of its code. Discuss the importance of obtaining such a statistical profile.

**2.12** What are the advantages and disadvantages of using the same system-call interface for manipulating both files and devices?

**2.13** Would it be possible for the user to develop a new command interpreter using the system-call interface provided by the operating system?

**2.14** Describe why Android uses ahead-of-time (AOT) rather than just-in-time (JIT) compilation.

**2.15** What are the two models of interprocess communication? What are the strengths and weaknesses of the two approaches?

**2.16** Contrast and compare an application programming interface (API) and an application binary interface (ABI).

**2.17** Why is the separation of mechanism and policy desirable?

**2.18** It is sometimes difficult to achieve a layered approach if two components of the operating system are dependent on each other. Identify a scenario in which it is unclear how to layer two system components that require tight coupling of their functionalities.

**2.19** What is the main advantage of the microkernel approach to system design? How do user programs and system services interact in a microkernel architecture? What are the disadvantages of using the microkernel approach?

**2.20** What are the advantages of using loadable kernel modules?

**2.21** How are iOS and Android similar? How are they different?

**2.22** Explain why Java programs running on Android systems do not use the standard Java API and virtual machine.

**2.23** The experimental Synthesis operating system has an assembler incorporated in the kernel. To optimize system-call performance, the kernel assembles routines within kernel space to minimize the path that the system call must take through the kernel. This approach is the antithesis of the layered approach, in which the path through the kernel is extended to make building the operating system easier. Discuss the pros and cons of the Synthesis approach to kernel design and system-performance optimization.

# Programming Problems

**2.24** In Section 2.3, we described a program that copies the contents of one file to a destination file. This program works by first prompting the user for the name of the source and destination files. Write this program using either the POSIX or Windows API. Be sure to include all necessary error checking, including ensuring that the source file exists.

Once you have correctly designed and tested the program, if you used a system that supports it, run the program using a utility that traces system calls. Linux systems provide the `strace` utility, and macOS systems use the `dtruss` command. (The `dtruss` command, which actually is a front end to `dtrace`, requires `admin` privileges, so it must be run using `sudo`.) These tools can be used as follows (assume that the name of the executable file is `FileCopy`:

**Linux:**

```
strace ./FileCopy
```

**macOS:**

```
sudo dtruss ./FileCopy
```

Since Windows systems do not provide such a tool, you will have to trace through the Windows version of this program using a debugger.

# Programming Projects

## Introduction to Linux Kernel Modules

In this project, you will learn how to create a kernel module and load it into the Linux kernel. You will then modify the kernel module so that it creates an entry in the `/proc` file system. The project can be completed using the Linux virtual machine that is available with this text. Although you may use any text editor to write these C programs, you will have to use the ***terminal*** application to compile the programs, and you will have to enter commands on the command line to manage the modules in the kernel.

As you'll discover, the advantage of developing kernel modules is that it is a relatively easy method of interacting with the kernel, thus allowing you to write programs that directly invoke kernel functions. It is important for you to keep in mind that you are indeed writing ***kernel code*** that directly interacts with the kernel. That normally means that any errors in the code could crash the system! However, since you will be using a virtual machine, any failures will at worst only require rebooting the system.

## I. Kernel Modules Overview

The first part of this project involves following a series of steps for creating and inserting a module into the Linux kernel.

You can list all kernel modules that are currently loaded by entering the command

```
lsmod
```

This command will list the current kernel modules in three columns: name, size, and where the module is being used.

```
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>

/* This function is called when the module is loaded. */
int simple_init(void)
{
   printk(KERN_INFO "Loading Kernel Module\n");

   return 0;
}

/* This function is called when the module is removed. */
void simple_exit(void)
{
   printk(KERN_INFO "Removing Kernel Module\n");
}

/* Macros for registering module entry and exit points. */
module_init(simple_init);
module_exit(simple_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Simple Module");
MODULE_AUTHOR("SGG");
```

**Figure 2.21** Kernel module `simple.c`.

The program in Figure 2.21 (named `simple.c` and available with the source code for this text) illustrates a very basic kernel module that prints appropriate messages when it is loaded and unloaded.

The function `simple_init()` is the **module entry point**, which represents the function that is invoked when the module is loaded into the kernel. Similarly, the `simple_exit()` function is the **module exit point**—the function that is called when the module is removed from the kernel.

The module entry point function must return an integer value, with 0 representing success and any other value representing failure. The module exit point function returns `void`. Neither the module entry point nor the module exit point is passed any parameters. The two following macros are used for registering the module entry and exit points with the kernel:

```
module_init(simple_init)

module_exit(simple_exit)
```

Notice in the figure how the module entry and exit point functions make calls to the `printk()` function. `printk()` is the kernel equivalent of `printf()`, but its output is sent to a kernel log buffer whose contents can be read by the `dmesg` command. One difference between `printf()` and `printk()` is that `printk()` allows us to specify a priority flag, whose values are given in the `<linux/printk.h>` include file. In this instance, the priority is `KERN_INFO`, which is defined as an ***informational*** message.

The final lines—`MODULE_LICENSE()`, `MODULE_DESCRIPTION()`, and `MODULE_AUTHOR()`—represent details regarding the software license, description of the module, and author. For our purposes, we do not require this information, but we include it because it is standard practice in developing kernel modules.

This kernel module `simple.c` is compiled using the `Makefile` accompanying the source code with this project. To compile the module, enter the following on the command line:

```
make
```

The compilation produces several files. The file `simple.ko` represents the compiled kernel module. The following step illustrates inserting this module into the Linux kernel.

## II. Loading and Removing Kernel Modules

Kernel modules are loaded using the `insmod` command, which is run as follows:

```
sudo insmod simple.ko
```

To check whether the module has loaded, enter the `lsmod` command and search for the module `simple`. Recall that the module entry point is invoked when the module is inserted into the kernel. To check the contents of this message in the kernel log buffer, enter the command

```
dmesg
```

You should see the message "`Loading Module.`"

Removing the kernel module involves invoking the `rmmod` command (notice that the `.ko` suffix is unnecessary):

```
sudo rmmod simple
```

Be sure to check with the `dmesg` command to ensure the module has been removed.

Because the kernel log buffer can fill up quickly, it often makes sense to clear the buffer periodically. This can be accomplished as follows:

```
sudo dmesg -c
```

Proceed through the steps described above to create the kernel module and to load and unload the module. Be sure to check the contents of the kernel log buffer using `dmesg` to ensure that you have followed the steps properly.

As kernel modules are running within the kernel, it is possible to obtain values and call functions that are available only in the kernel and not to regular user applications. For example, the Linux include file `<linux/hash.h>` defines several hashing functions for use within the kernel. This file also defines the constant value `GOLDEN_RATIO_PRIME` (which is defined as an `unsigned long`). This value can be printed out as follows:

```
printk(KERN_INFO "%lu\n", GOLDEN_RATIO_PRIME);
```

As another example, the include file `<linux/gcd.h>` defines the following function

```
unsigned long gcd(unsigned long a, unsigned b);
```

which returns the greatest common divisor of the parameters `a` and `b`.

Once you are able to correctly load and unload your module, complete the following additional steps:

1. Print out the value of `GOLDEN_RATIO_PRIME` in the `simple_init()` function.
2. Print out the greatest common divisor of 3,300 and 24 in the `simple_exit()` function.

As compiler errors are not often helpful when performing kernel development, it is important to compile your program often by running `make` regularly. Be sure to load and remove the kernel module and check the kernel log buffer using `dmesg` to ensure that your changes to `simple.c` are working properly.

In Section 1.4.3, we described the role of the timer as well as the timer interrupt handler. In Linux, the rate at which the timer ticks (the **tick rate**) is the value `HZ` defined in `<asm/param.h>`. The value of `HZ` determines the frequency of the timer interrupt, and its value varies by machine type and architecture. For example, if the value of `HZ` is 100, a timer interrupt occurs 100 times per second, or every 10 milliseconds. Additionally, the kernel keeps track of the global variable `jiffies`, which maintains the number of timer interrupts that have occurred since the system was booted. The `jiffies` variable is declared in the file `<linux/jiffies.h>`.

1. Print out the values of `jiffies` and `HZ` in the `simple_init()` function.
2. Print out the value of `jiffies` in the `simple_exit()` function.

Before proceeding to the next set of exercises, consider how you can use the different values of `jiffies` in `simple_init()` and `simple_exit()` to determine the number of seconds that have elapsed since the time the kernel module was loaded and then removed.

## III. The `/proc` File System

The `/proc` file system is a "pseudo" file system that exists only in kernel memory and is used primarily for querying various kernel and per-process statistics.

```
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <asm/uaccess.h>

#define BUFFER_SIZE 128
#define PROC_NAME "hello"

ssize_t proc_read(struct file *file, char __user *usr_buf,
  size_t count, loff_t *pos);

static struct file_operations proc_ops = {
    .owner = THIS_MODULE,
    .read = proc_read,
};

/* This function is called when the module is loaded. */
int proc_init(void)
{
  /* creates the /proc/hello entry */
  proc_create(PROC_NAME, 0666, NULL, &proc_ops);

  return 0;
}

/* This function is called when the module is removed. */
void proc_exit(void)
{
  /* removes the /proc/hello entry */
  remove_proc_entry(PROC_NAME, NULL);
}
```

**Figure 2.22** The `/proc` file-system kernel module, Part 1

This exercise involves designing kernel modules that create additional entries in the `/proc` file system involving both kernel statistics and information related to specific processes. The entire program is included in Figure 2.22 and Figure 2.23.

We begin by describing how to create a new entry in the `/proc` file system. The following program example (named `hello.c` and available with the source code for this text) creates a `/proc` entry named `/proc/hello`. If a user enters the command

```
cat /proc/hello
```

the infamous `Hello World` message is returned.

```
/* This function is called each time /proc/hello is read */
ssize_t proc_read(struct file *file, char __user *usr_buf,
  size_t count, loff_t *pos)
{
  int rv = 0;
  char buffer[BUFFER_SIZE];
  static int completed = 0;

  if (completed) {
   completed = 0;
   return 0;
  }

  completed = 1;

  rv = sprintf(buffer, "Hello World\n");

  /* copies kernel space buffer to user space usr_buf */
  copy_to_user(usr_buf, buffer, rv);

  return rv;
}
module_init(proc_init);
module_exit(proc_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Hello Module");
MODULE_AUTHOR("SGG");
```

**Figure 2.23** The `/proc` file system kernel module, Part 2

In the module entry point `proc_init()`, we create the new `/proc/hello` entry using the `proc_create()` function. This function is passed `proc_ops`, which contains a reference to a `struct file_operations`. This struct initializes the `.owner` and `.read` members. The value of `.read` is the name of the function `proc_read()` that is to be called whenever `/proc/hello` is read.

Examining this `proc_read()` function, we see that the string `"Hello World\n"` is written to the variable `buffer` where `buffer` exists in kernel memory. Since `/proc/hello` can be accessed from user space, we must copy the contents of `buffer` to user space using the kernel function `copy_to_user()`. This function copies the contents of kernel memory `buffer` to the variable `usr_buf`, which exists in user space.

Each time the `/proc/hello` file is read, the `proc_read()` function is called repeatedly until it returns 0, so there must be logic to ensure that this function returns 0 once it has collected the data (in this case, the string `"Hello World\n"`) that is to go into the corresponding `/proc/hello` file.

Finally, notice that the `/proc/hello` file is removed in the module exit point `proc_exit()` using the function `remove_proc_entry()`.

## IV. Assignment

This assignment will involve designing two kernel modules:

1. Design a kernel module that creates a `/proc` file named `/proc/jiffies` that reports the current value of `jiffies` when the `/proc/jiffies` file is read, such as with the command

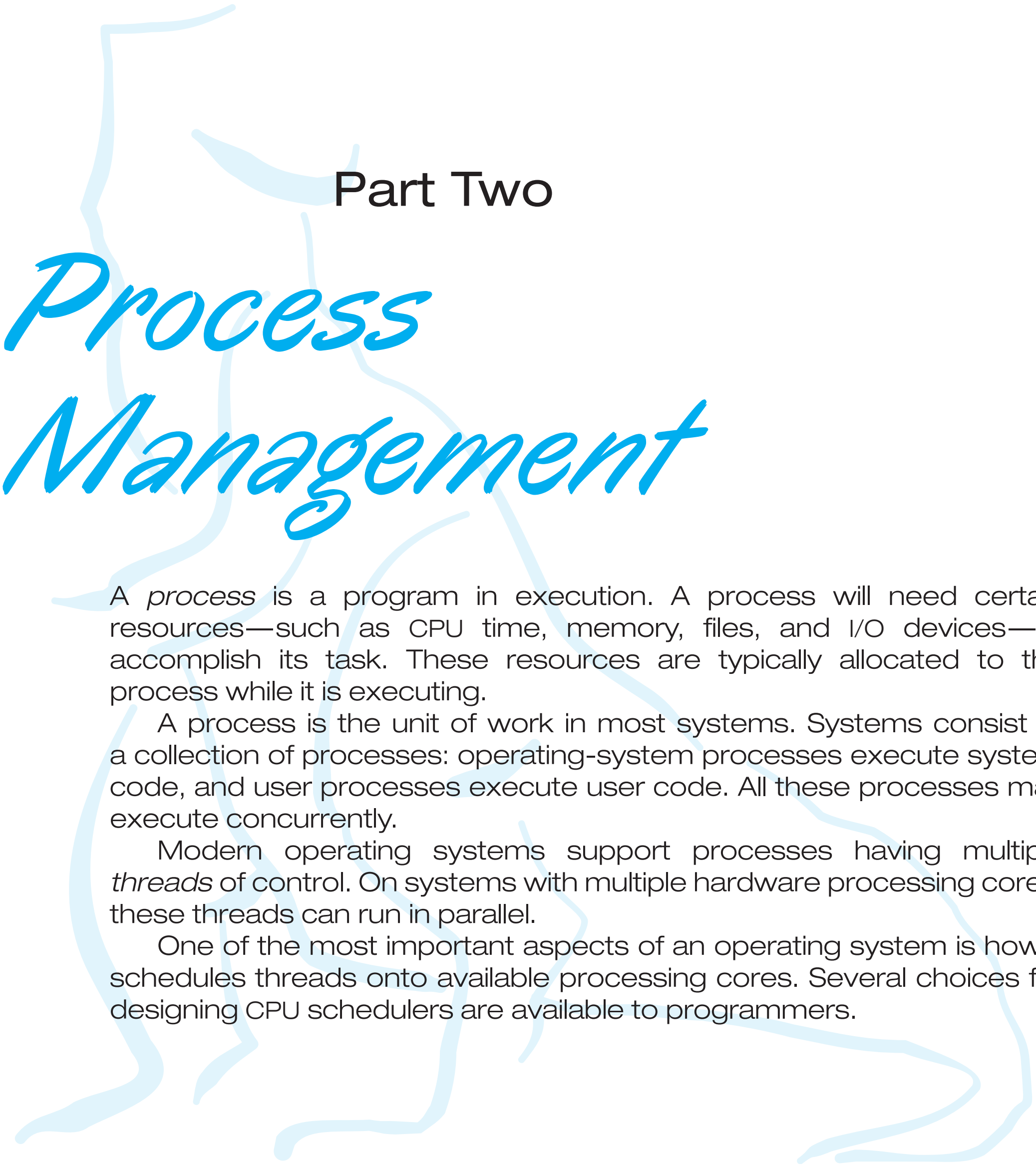
   ```
   cat /proc/jiffies
   ```

   Be sure to remove `/proc/jiffies` when the module is removed.

2. Design a kernel module that creates a `proc` file named `/proc/seconds` that reports the number of elapsed seconds since the kernel module was loaded. This will involve using the value of `jiffies` as well as the `HZ` rate. When a user enters the command

   ```
   cat /proc/seconds
   ```

   your kernel module will report the number of seconds that have elapsed since the kernel module was first loaded. Be sure to remove `/proc/seconds` when the module is removed.



# Part Two

# Process Management

A *process* is a program in execution. A process will need certain resources—such as CPU time, memory, files, and I/O devices—to accomplish its task. These resources are typically allocated to the process while it is executing.

A process is the unit of work in most systems. Systems consist of a collection of processes: operating-system processes execute system code, and user processes execute user code. All these processes may execute concurrently.

Modern operating systems support processes having multiple *threads* of control. On systems with multiple hardware processing cores, these threads can run in parallel.

One of the most important aspects of an operating system is how it schedules threads onto available processing cores. Several choices for designing CPU schedulers are available to programmers.

CHAPTER 3

# Processes

Early computers allowed only one program to be executed at a time. This program had complete control of the system and had access to all the system's resources. In contrast, contemporary computer systems allow multiple programs to be loaded into memory and executed concurrently. This evolution required firmer control and more compartmentalization of the various programs; and these needs resulted in the notion of a **process**, which is a program in execution. A process is the unit of work in a modern computing system.

The more complex the operating system is, the more it is expected to do on behalf of its users. Although its main concern is the execution of user programs, it also needs to take care of various system tasks that are best done in user space, rather than within the kernel. A system therefore consists of a collection of processes, some executing user code, others executing operating system code. Potentially, all these processes can execute concurrently, with the CPU (or CPUs) multiplexed among them. In this chapter, you will read about what processes are, how they are represented in an operating system, and how they work.

## CHAPTER OBJECTIVES

- Identify the separate components of a process and illustrate how they are represented and scheduled in an operating system.
- Describe how processes are created and terminated in an operating system, including developing programs using the appropriate system calls that perform these operations.
- Describe and contrast interprocess communication using shared memory and message passing.
- Design programs that use pipes and POSIX shared memory to perform interprocess communication.
- Describe client–server communication using sockets and remote procedure calls.
- Design kernel modules that interact with the Linux operating system.

## 3.1 Process Concept

A question that arises in discussing operating systems involves what to call all the CPU activities. Early computers were batch systems that executed **jobs**, followed by the emergence of time-shared systems that ran **user programs**, or **tasks**. Even on a single-user system, a user may be able to run several programs at one time: a word processor, a web browser, and an e-mail package. And even if a computer can execute only one program at a time, such as on an embedded device that does not support multitasking, the operating system may need to support its own internal programmed activities, such as memory management. In many respects, all these activities are similar, so we call all of them **processes**.

Although we personally prefer the more contemporary term ***process***, the term ***job*** has historical significance, as much of operating system theory and terminology was developed during a time when the major activity of operating systems was job processing. Therefore, in some appropriate instances we use ***job*** when describing the role of the operating system. As an example, it would be misleading to avoid the use of commonly accepted terms that include the word ***job*** (such as ***job scheduling***) simply because ***process*** has superseded ***job***.

### 3.1.1 The Process

Informally, as mentioned earlier, a process is a program in execution. The status of the current activity of a process is represented by the value of the **program counter** and the contents of the processor's registers. The memory layout of a process is typically divided into multiple sections, and is shown in Figure 3.1. These sections include:

- **Text section**—the executable code
- **Data section**—global variables

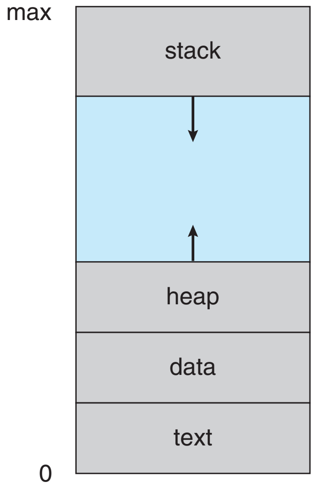


**Figure 3.1** Layout of a process in memory.

- **Heap section**—memory that is dynamically allocated during program run time
- **Stack section**—temporary data storage when invoking functions (such as function parameters, return addresses, and local variables)

Notice that the sizes of the text and data sections are fixed, as their sizes do not change during program run time. However, the stack and heap sections can shrink and grow dynamically during program execution. Each time a function is called, an **activation record** containing function parameters, local variables, and the return address is pushed onto the stack; when control is returned from the function, the activation record is popped from the stack. Similarly, the heap will grow as memory is dynamically allocated, and will shrink when memory is returned to the system. Although the stack and heap sections grow ***toward*** one another, the operating system must ensure they do not ***overlap*** one another.

We emphasize that a program by itself is not a process. A program is a ***passive*** entity, such as a file containing a list of instructions stored on disk (often called an **executable fil** ). In contrast, a process is an ***active*** entity, with a program counter specifying the next instruction to execute and a set of associated resources. A program becomes a process when an executable file is loaded into memory. Two common techniques for loading executable files are double-clicking an icon representing the executable file and entering the name of the executable file on the command line (as in `prog.exe` or `a.out`).

Although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. For instance, several users may be running different copies of the mail program, or the same user may invoke many copies of the web browser program. Each of these is a separate process; and although the text sections are equivalent, the data, heap, and stack sections vary. It is also common to have a process that spawns many processes as it runs. We discuss such matters in Section 3.4.

Note that a process can itself be an execution environment for other code. The Java programming environment provides a good example. In most circumstances, an executable Java program is executed within the Java virtual machine (JVM). The JVM executes as a process that interprets the loaded Java code and takes actions (via native machine instructions) on behalf of that code. For example, to run the compiled Java program `Program.class`, we would enter

```
java Program
```

The command `java` runs the JVM as an ordinary process, which in turns executes the Java program `Program` in the virtual machine. The concept is the same as simulation, except that the code, instead of being written for a different instruction set, is written in the Java language.

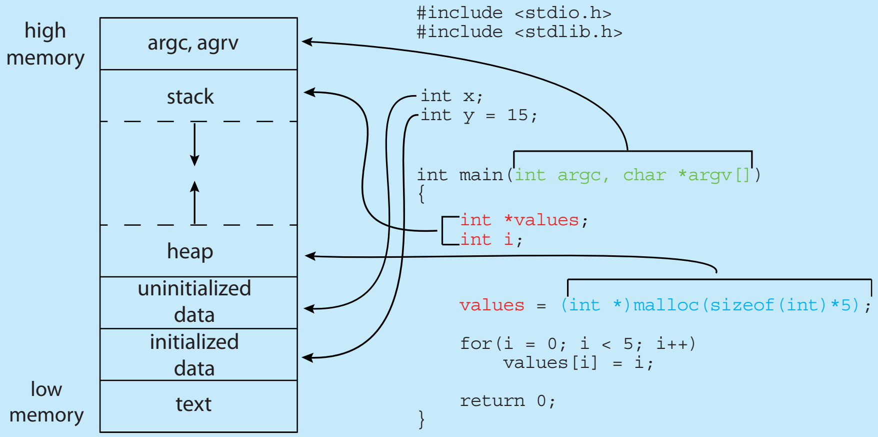
### 3.1.2 Process State

As a process executes, it changes **state**. The state of a process is defined in part by the current activity of that process. A process may be in one of the following states:

### *MEMORY LAYOUT OF A C PROGRAM*

The figure shown below illustrates the layout of a C program in memory, highlighting how the different sections of a process relate to an actual C program. This figure is similar to the general concept of a process in memory as shown in Figure 3.1, with a few differences:

- The global data section is divided into different sections for (a) initialized data and (b) uninitialized data.
- A separate section is provided for the `argc` and `argv` parameters passed to the `main()` function.



```
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 15;

int main(int argc, char *argv[])
{
      int *values;
      int i;

      values = (int *)malloc(sizeof(int)*5);

      for(i = 0; i < 5; i++)
          values[i] = i;

      return 0;
}
```

The GNU `size` command can be used to determine the size (in bytes) of some of these sections. Assuming the name of the executable file of the above C program is `memory`, the following is the output generated by entering the command `size memory`:

```
text    data    bss     dec     hex     filename
1158    284     8       1450    5aa     memory
```

The `data` field refers to uninitialized data, and `bss` refers to initialized data. (`bss` is a historical term referring to ***block started by symbol***.) The `dec` and `hex` values are the sum of the three sections represented in decimal and hexadecimal, respectively.

- **New**. The process is being created.
- **Running**. Instructions are being executed.
- **Waiting**. The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- **Ready**. The process is waiting to be assigned to a processor.

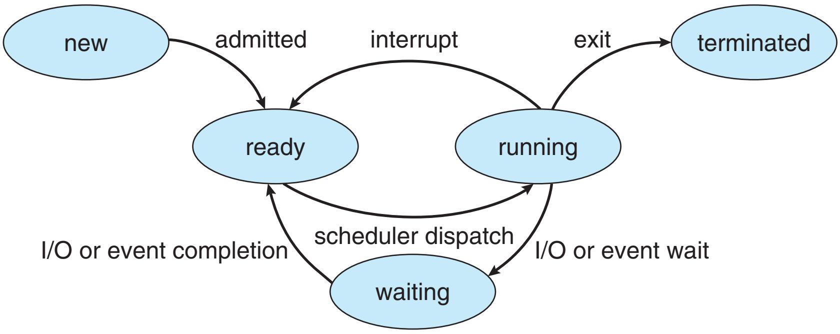

new
admitted
interrupt
exit
terminated
ready
running
scheduler dispatch
I/O or event completion
I/O or event wait
waiting

**Figure 3.2** Diagram of process state.

- **Terminated**. The process has finished execution.

These names are arbitrary, and they vary across operating systems. The states that they represent are found on all systems, however. Certain operating systems also more finely delineate process states. It is important to realize that only one process can be *running* on any processor core at any instant. Many processes may be *ready* and *waiting*, however. The state diagram corresponding to these states is presented in Figure 3.2.

### 3.1.3 Process Control Block

Each process is represented in the operating system by a **process control block** (PCB)—also called a **task control block**. A PCB is shown in Figure 3.3. It contains many pieces of information associated with a specific process, including these:

- **Process state**. The state may be new, ready, running, waiting, halted, and so on.
- **Program counter**. The counter indicates the address of the next instruction to be executed for this process.

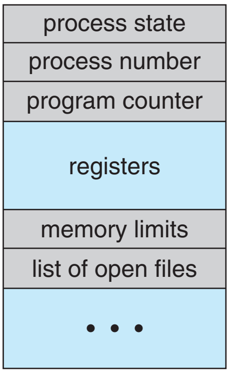

process state
process number
program counter
registers
memory limits
list of open files
• • •

**Figure 3.3** Process control block (PCB).

- **CPU registers**. The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward when it is rescheduled to run.
- **CPU-scheduling information**. This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters. (Chapter 5 describes process scheduling.)
- **Memory-management information**. This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system (Chapter 9).
- **Accounting information**. This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- **I/O status information**. This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

In brief, the PCB simply serves as the repository for all the data needed to start, or restart, a process, along with some accounting data.

### 3.1.4 Threads

The process model discussed so far has implied that a process is a program that performs a single **thread** of execution. For example, when a process is running a word-processor program, a single thread of instructions is being executed. This single thread of control allows the process to perform only one task at a time. Thus, the user cannot simultaneously type in characters and run the spell checker. Most modern operating systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time. This feature is especially beneficial on multicore systems, where multiple threads can run in parallel. A multithreaded word processor could, for example, assign one thread to manage user input while another thread runs the spell checker. On systems that support threads, the PCB is expanded to include information for each thread. Other changes throughout the system are also needed to support threads. Chapter 4 explores threads in detail.

## 3.2 Process Scheduling

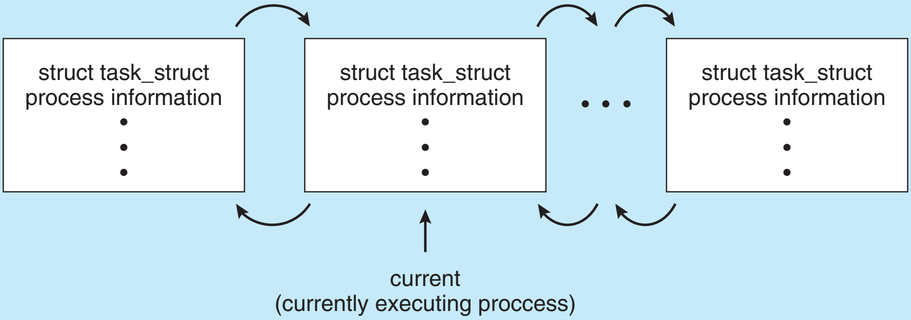
The objective of multiprogramming is to have some process running at all times so as to maximize CPU utilization. The objective of time sharing is to switch a CPU core among processes so frequently that users can interact with each program while it is running. To meet these objectives, the **process scheduler** selects an available process (possibly from a set of several available processes) for program execution on a core. Each CPU core can run one process at a time.

### *PROCESS REPRESENTATION IN LINUX*

The process control block in the Linux operating system is represented by the C structure `task_struct`, which is found in the `<include/linux/sched.h>` include file in the kernel source-code directory. This structure contains all the necessary information for representing a process, including the state of the process, scheduling and memory-management information, list of open files, and pointers to the process's parent and a list of its children and siblings. (A process's **parent** is the process that created it; its **children** are any processes that it creates. Its **siblings** are children with the same parent process.) Some of these fields include:

```
long state;                    /* state of the process */
struct sched_entity se;        /* scheduling information */
struct task_struct *parent;    /* this process's parent */
struct list_head children;     /* this process's children */
struct files_struct *files;    /* list of open files */
struct mm_struct *mm;          /* address space */
```

For example, the state of a process is represented by the field `long state` in this structure. Within the Linux kernel, all active processes are represented using a doubly linked list of `task_struct`. The kernel maintains a pointer–`current`–to the process currently executing on the system, as shown below:



As an illustration of how the kernel might manipulate one of the fields in the `task_struct` for a specified process, let's assume the system would like to change the state of the process currently running to the value `new_state`. If `current` is a pointer to the process currently executing, its state is changed with the following:

```
current->state = new_state;
```

For a system with a single CPU core, there will never be more than one process running at a time, whereas a multicore system can run multiple processes at one time. If there are more processes than cores, excess processes will have

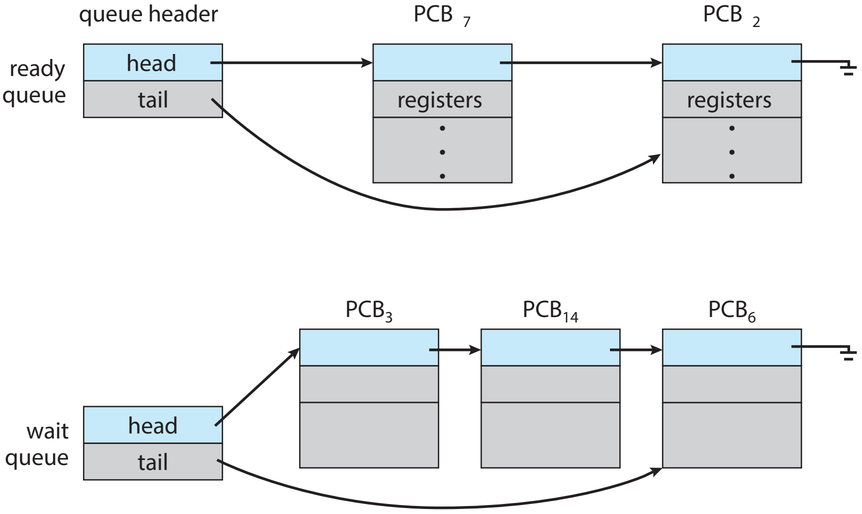


**Figure 3.4** The ready queue and wait queues.

to wait until a core is free and can be rescheduled. The number of processes currently in memory is known as the **degree of multiprogramming**.

Balancing the objectives of multiprogramming and time sharing also requires taking the general behavior of a process into account. In general, most processes can be described as either I/O bound or CPU bound. An **I/O-bound process** is one that spends more of its time doing I/O than it spends doing computations. A **CPU-bound process**, in contrast, generates I/O requests infrequently, using more of its time doing computations.

### 3.2.1 Scheduling Queues

As processes enter the system, they are put into a **ready queue**, where they are ready and waiting to execute on a CPU's core This queue is generally stored as a linked list; a ready-queue header contains pointers to the first PCB in the list, and each PCB includes a pointer field that points to the next PCB in the ready queue.

The system also includes other queues. When a process is allocated a CPU core, it executes for a while and eventually terminates, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request. Suppose the process makes an I/O request to a device such as a disk. Since devices run significantly slower than processors, the process will have to wait for the I/O to become available. Processes that are waiting for a certain event to occur — such as completion of I/O — are placed in a **wait queue** (Figure 3.4).

A common representation of process scheduling is a **queueing diagram**, such as that in Figure 3.5. Two types of queues are present: the ready queue and a set of wait queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

A new process is initially put in the ready queue. It waits there until it is selected for execution, or **dispatched**. Once the process is allocated a CPU core and is executing, one of several events could occur:

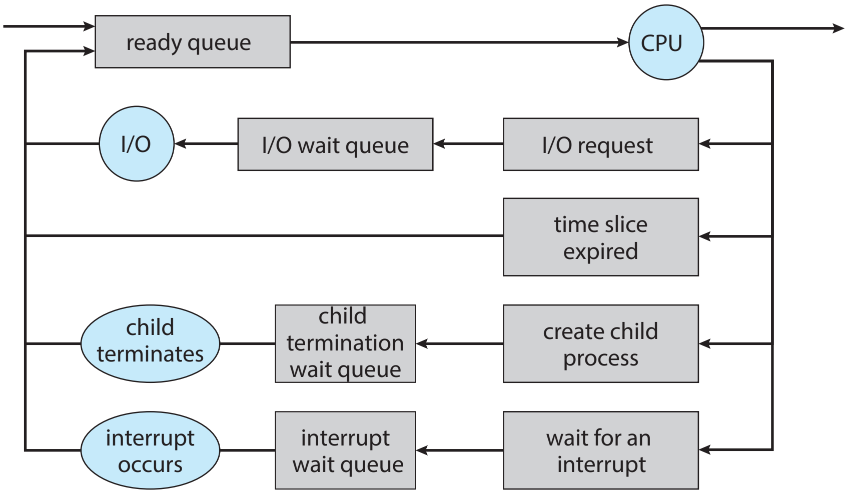


**Figure 3.5** Queueing-diagram representation of process scheduling.

- The process could issue an I/O request and then be placed in an I/O wait queue.
- The process could create a new child process and then be placed in a wait queue while it awaits the child's termination.
- The process could be removed forcibly from the core, as a result of an interrupt or having its time slice expire, and be put back in the ready queue.

In the first two cases, the process eventually switches from the waiting state to the ready state and is then put back in the ready queue. A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

### 3.2.2 CPU Scheduling

A process migrates among the ready queue and various wait queues throughout its lifetime. The role of the **CPU scheduler** is to select from among the processes that are in the ready queue and allocate a CPU core to one of them. The CPU scheduler must select a new process for the CPU frequently. An I/O-bound process may execute for only a few milliseconds before waiting for an I/O request. Although a CPU-bound process will require a CPU core for longer durations, the scheduler is unlikely to grant the core to a process for an extended period. Instead, it is likely designed to forcibly remove the CPU from a process and schedule another process to run. Therefore, the CPU scheduler executes at least once every 100 milliseconds, although typically much more frequently.

Some operating systems have an intermediate form of scheduling, known as **swapping**, whose key idea is that sometimes it can be advantageous to remove a process from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramming. Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is known as ***swapping*** because a process can be "swapped out" from memory to disk, where its current status is saved, and later "swapped in" from disk back to memory, where its status is restored. Swapping is typically only necessary when memory has been overcommitted and must be freed up. Swapping is discussed in Chapter 9.

### 3.2.3 Context Switch

As mentioned in Section 1.2.1, interrupts cause the operating system to change a CPU core from its current task and to run a kernel routine. Such operations happen frequently on general-purpose systems. When an interrupt occurs, the system needs to save the current **context** of the process running on the CPU core so that it can restore that context when its processing is done, essentially suspending the process and then resuming it. The context is represented in the PCB of the process. It includes the value of the CPU registers, the process state (see Figure 3.2), and memory-management information. Generically, we perform a **state save** of the current state of the CPU core, be it in kernel or user mode, and then a **state restore** to resume operations.

Switching the CPU core to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a **context switch** and is illustrated in Figure 3.6. When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run. Context-switch time is pure overhead, because the system does no useful work while switching. Switching speed varies from machine to machine, depending on the

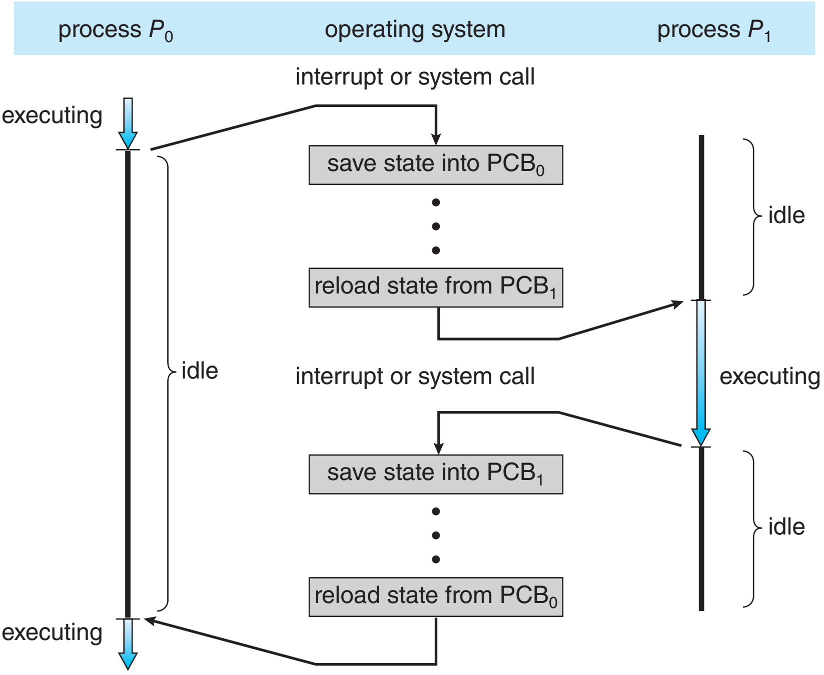

process $P_0$
operating system
process $P_1$
interrupt or system call
executing
save state into $PCB_0$
idle
reload state from $PCB_1$
idle
interrupt or system call
executing
save state into $PCB_1$
idle
reload state from $PCB_0$
executing

Figure 3.6 Diagram showing context switch from process to process.

### *MULTITASKING IN MOBILE SYSTEMS*

Because of the constraints imposed on mobile devices, early versions of iOS did not provide user-application multitasking; only one application ran in the foreground while all other user applications were suspended. Operating-system tasks were multitasked because they were written by Apple and well behaved. However, beginning with iOS 4, Apple provided a limited form of multitasking for user applications, thus allowing a single foreground application to run concurrently with multiple background applications. (On a mobile device, the **foreground** application is the application currently open and appearing on the display. The **background** application remains in memory, but does not occupy the display screen.) The iOS 4 programming API provided support for multitasking, thus allowing a process to run in the background without being suspended. However, it was limited and only available for a few application types. As hardware for mobile devices began to offer larger memory capacities, multiple processing cores, and greater battery life, subsequent versions of iOS began to support richer functionality for multitasking with fewer restrictions. For example, the larger screen on iPad tablets allowed running two foreground apps at the same time, a technique known as **split-screen**.

Since its origins, Android has supported multitasking and does not place constraints on the types of applications that can run in the background. If an application requires processing while in the background, the application must use a **service**, a separate application component that runs on behalf of the background process. Consider a streaming audio application: if the application moves to the background, the service continues to send audio data to the audio device driver on behalf of the background application. In fact, the service will continue to run even if the background application is suspended. Services do not have a user interface and have a small memory footprint, thus providing an efficient technique for multitasking in a mobile environment.

memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers). A typical speed is a several microseconds.

Context-switch times are highly dependent on hardware support. For instance, some processors provide multiple sets of registers. A context switch here simply requires changing the pointer to the current register set. Of course, if there are more active processes than there are register sets, the system resorts to copying register data to and from memory, as before. Also, the more complex the operating system, the greater the amount of work that must be done during a context switch. As we will see in Chapter 9, advanced memory-management techniques may require that extra data be switched with each context. For instance, the address space of the current process must be preserved as the space of the next task is prepared for use. How the address space is preserved, and what amount of work is needed to preserve it, depend on the memory-management method of the operating system.

## 3.3 Operations on Processes

The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination. In this section, we explore the mechanisms involved in creating processes and illustrate process creation on UNIX and Windows systems.

### 3.3.1 Process Creation

During the course of execution, a process may create several new processes. As mentioned earlier, the creating process is called a parent process, and the new processes are called the children of that process. Each of these new processes may in turn create other processes, forming a **tree** of processes.

Most operating systems (including UNIX, Linux, and Windows) identify processes according to a unique **process identifier** (or **pid**), which is typically an integer number. The pid provides a unique value for each process in the system, and it can be used as an index to access various attributes of a process within the kernel.

Figure 3.7 illustrates a typical process tree for the Linux operating system, showing the name of each process and its pid. (We use the term ***process*** rather loosely in this situation, as Linux prefers the term ***task*** instead.) The `systemd` process (which always has a pid of 1) serves as the root parent process for all user processes, and is the first user process created when the system boots. Once the system has booted, the `systemd` process creates processes which provide additional services such as a web or print server, an `ssh` server, and the like. In Figure 3.7, we see two children of `systemd`—`logind` and `sshd`. The `logind` process is responsible for managing clients that directly log onto the system. In this example, a client has logged on and is using the `bash` shell, which has been assigned pid 8416. Using the `bash` command-line interface, this user has created the process `ps` as well as the `vim` editor. The `sshd` process is responsible for managing clients that connect to the system by using `ssh` (which is short for ***secure shell***).

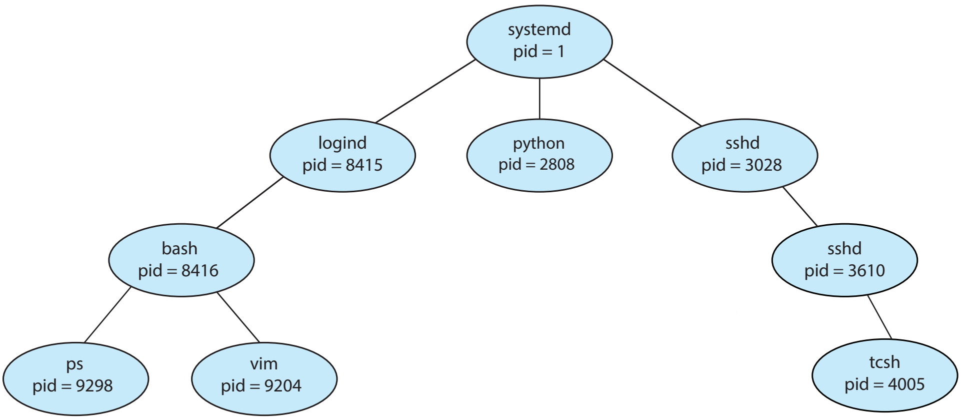


**Figure 3.7** A tree of processes on a typical Linux system.

### THE *init* AND *systemd* PROCESSES

Traditional UNIX systems identify the process `init` as the root of all child processes. `init` (also known as **System V `init`**) is assigned a pid of 1, and is the first process created when the system is booted. On a process tree similar to what is shown in Figure 3.7, `init` is at the root.

Linux systems initially adopted the System V `init` approach, but recent distributions have replaced it with `systemd`. As described in Section 3.3.1, `systemd` serves as the system's initial process, much the same as System V `init`; however it is much more flexible, and can provide more services, than `init`.

On UNIX and Linux systems, we can obtain a listing of processes by using the `ps` command. For example, the command

```
ps -el
```

will list complete information for all processes currently active in the system. A process tree similar to the one shown in Figure 3.7 can be constructed by recursively tracing parent processes all the way to the `systemd` process. (In addition, Linux systems provide the `pstree` command, which displays a tree of all processes in the system.)

In general, when a process creates a child process, that child process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task. A child process may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process. The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children. Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many child processes.

In addition to supplying various physical and logical resources, the parent process may pass along initialization data (input) to the child process. For example, consider a process whose function is to display the contents of a file—say, `hw1.c`—on the screen of a terminal. When the process is created, it will get, as an input from its parent process, the name of the file `hw1.c`. Using that file name, it will open the file and write the contents out. It may also get the name of the output device. Alternatively, some operating systems pass resources to child processes. On such a system, the new process may get two open files, `hw1.c` and the terminal device, and may simply transfer the datum between the two.

When a process creates a new process, two possibilities for execution exist:

1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated.

There are also two address-space possibilities for the new process:

1. The child process is a duplicate of the parent process (it has the same program and data as the parent).
2. The child process has a new program loaded into it.

To illustrate these differences, let's first consider the UNIX operating system. In UNIX, as we've seen, each process is identified by its process identifier, which is a unique integer. A new process is created by the `fork()` system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process. Both processes (the parent and the child) continue execution at the instruction after the `fork()`, with one difference: the return code for the `fork()` is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

After a `fork()` system call, one of the two processes typically uses the `exec()` system call to replace the process's memory space with a new program. The `exec()` system call loads a binary file into memory (destroying the memory image of the program containing the `exec()` system call) and starts

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
    }
    else if (pid == 0) { /* child process */
      execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete");
    }

    return 0;
}
```

**Figure 3.8** Creating a separate process using the UNIX `fork()` system call.

its execution. In this manner, the two processes are able to communicate and then go their separate ways. The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue a `wait()` system call to move itself off the ready queue until the termination of the child. Because the call to `exec()` overlays the process's address space with a new program, `exec()` does not return control unless an error occurs.

The C program shown in Figure 3.8 illustrates the UNIX system calls previously described. We now have two different processes running copies of the same program. The only difference is that the value of the variable `pid` for the child process is zero, while that for the parent is an integer value greater than zero (in fact, it is the actual pid of the child process). The child process inherits privileges and scheduling attributes from the parent, as well certain resources, such as open files. The child process then overlays its address space with the UNIX command `/bin/ls` (used to get a directory listing) using the `execlp()` system call (`execlp()` is a version of the `exec()` system call). The parent waits for the child process to complete with the `wait()` system call. When the child process completes (by either implicitly or explicitly invoking `exit()`), the parent process resumes from the call to `wait()`, where it completes using the `exit()` system call. This is also illustrated in Figure 3.9.

Of course, there is nothing to prevent the child from ***not*** invoking `exec()` and instead continuing to execute as a copy of the parent process. In this scenario, the parent and child are concurrent processes running the same code instructions. Because the child is a copy of the parent, each process has its own copy of any data.

As an alternative example, we next consider process creation in Windows. Processes are created in the Windows API using the `CreateProcess()` function, which is similar to `fork()` in that a parent creates a new child process. However, whereas `fork()` has the child process inheriting the address space of its parent, `CreateProcess()` requires loading a specified program into the address space of the child process at process creation. Furthermore, whereas `fork()` is passed no parameters, `CreateProcess()` expects no fewer than ten parameters.

The C program shown in Figure 3.10 illustrates the `CreateProcess()` function, which creates a child process that loads the application `mspaint.exe`. We opt for many of the default values of the ten parameters passed to `CreateProcess()`. Readers interested in pursuing the details of process creation and management in the Windows API are encouraged to consult the bibliographical notes at the end of this chapter.

The two parameters passed to the `CreateProcess()` function are instances of the `STARTUPINFO` and `PROCESS_INFORMATION` structures. `STARTUPINFO` specifies many properties of the new process, such as window

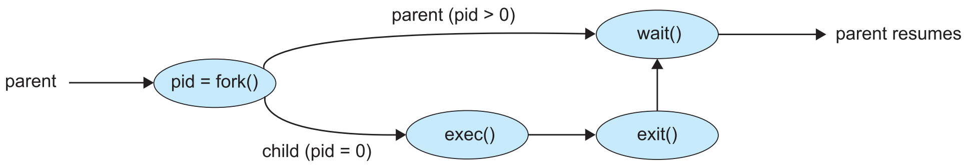


**Figure 3.9** Process creation using the `fork()` system call.

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
     "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
     NULL, /* don't inherit process handle */
     NULL, /* don't inherit thread handle */
     FALSE, /* disable handle inheritance */
     0, /* no creation flags */
     NULL, /* use parent's environment block */
     NULL, /* use parent's existing directory */
     &si,
     &pi))
    {
      fprintf(stderr, "Create Process Failed");
      return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

**Figure 3.10** Creating a separate process using the Windows API.

size and appearance and handles to standard input and output files. The `PROCESS_INFORMATION` structure contains a handle and the identifiers to the newly created process and its thread. We invoke the `ZeroMemory()` function to allocate memory for each of these structures before proceeding with `CreateProcess()`.

The first two parameters passed to `CreateProcess()` are the application name and command-line parameters. If the application name is NULL (as it is in this case), the command-line parameter specifies the application to load.

In this instance, we are loading the Microsoft Windows `mspaint.exe` application. Beyond these two initial parameters, we use the default parameters for inheriting process and thread handles as well as specifying that there will be no creation flags. We also use the parent's existing environment block and starting directory. Last, we provide two pointers to the `STARTUPINFO` and `PROCESS_INFORMATION` structures created at the beginning of the program. In Figure 3.8, the parent process waits for the child to complete by invoking the `wait()` system call. The equivalent of this in Windows is `WaitForSingleObject()`, which is passed a handle of the child process—`pi.hProcess`—and waits for this process to complete. Once the child process exits, control returns from the `WaitForSingleObject()` function in the parent process.

### 3.3.2 Process Termination

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the `exit()` system call. At that point, the process may return a status value (typically an integer) to its waiting parent process (via the `wait()` system call). All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated and reclaimed by the operating system.

Termination can occur in other circumstances as well. A process can cause the termination of another process via an appropriate system call (for example, `TerminateProcess()` in Windows). Usually, such a system call can be invoked only by the parent of the process that is to be terminated. Otherwise, a user—or a misbehaving application—could arbitrarily kill another user's processes. Note that a parent needs to know the identities of its children if it is to terminate them. Thus, when one process creates a new process, the identity of the newly created process is passed to the parent.

A parent may terminate the execution of one of its children for a variety of reasons, such as these:

- The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)
- The task assigned to the child is no longer required.
- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

Some systems do not allow a child to exist if its parent has terminated. In such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as **cascading termination**, is normally initiated by the operating system.

To illustrate process execution and termination, consider that, in Linux and UNIX systems, we can terminate a process by using the `exit()` system call, providing an exit status as a parameter:

```
/* exit with status 1 */
exit(1);
```

In fact, under normal termination, `exit()` will be called either directly (as shown above) or indirectly, as the C run-time library (which is added to UNIX executable files) will include a call to `exit()` by default.

A parent process may wait for the termination of a child process by using the `wait()` system call. The `wait()` system call is passed a parameter that allows the parent to obtain the exit status of the child. This system call also returns the process identifier of the terminated child so that the parent can tell which of its children has terminated:

```
pid_t pid;
int status;

pid = wait(&status);
```

When a process terminates, its resources are deallocated by the operating system. However, its entry in the process table must remain there until the parent calls `wait()`, because the process table contains the process's exit status. A process that has terminated, but whose parent has not yet called `wait()`, is known as a **zombie** process. All processes transition to this state when they terminate, but generally they exist as zombies only briefly. Once the parent calls `wait()`, the process identifier of the zombie process and its entry in the process table are released.

Now consider what would happen if a parent did not invoke `wait()` and instead terminated, thereby leaving its child processes as **orphans**. Traditional UNIX systems addressed this scenario by assigning the `init` process as the new parent to orphan processes. (Recall from Section 3.3.1 that `init` serves as the root of the process hierarchy in UNIX systems.) The `init` process periodically invokes `wait()`, thereby allowing the exit status of any orphaned process to be collected and releasing the orphan's process identifier and process-table entry.

Although most Linux systems have replaced `init` with `systemd`, the latter process can still serve the same role, although Linux also allows processes other than `systemd` to inherit orphan processes and manage their termination.

### 3.3.2.1 Android Process Hierarchy

Because of resource constraints such as limited memory, mobile operating systems may have to terminate existing processes to reclaim limited system resources. Rather than terminating an arbitrary process, Android has identified an ***importance hierarchy*** of processes, and when the system must terminate a process to make resources available for a new, or more important, process, it terminates processes in order of increasing importance. From most to least important, the hierarchy of process classifications is as follows:

- **Foreground process**—The current process visible on the screen, representing the application the user is currently interacting with
- **Visible process**—A process that is not directly visible on the foreground but that is performing an activity that the foreground process is referring to (that is, a process performing an activity whose status is displayed on the foreground process)

- **Service process**—A process that is similar to a background process but is performing an activity that is apparent to the user (such as streaming music)
- **Background process**—A process that may be performing an activity but is not apparent to the user.
- **Empty process**—A process that holds no active components associated with any application

If system resources must be reclaimed, Android will first terminate empty processes, followed by background processes, and so forth. Processes are assigned an importance ranking, and Android attempts to assign a process as high a ranking as possible. For example, if a process is providing a service and is also visible, it will be assigned the more-important visible classification.

Furthermore, Android development practices suggest following the guidelines of the process life cycle. When these guidelines are followed, the state of a process will be saved prior to termination and resumed at its saved state if the user navigates back to the application.

## 3.4 Interprocess Communication

Processes executing concurrently in the operating system may be either independent processes or cooperating processes. A process is ***independent*** if it does not share data with any other processes executing in the system. A process is ***cooperating*** if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

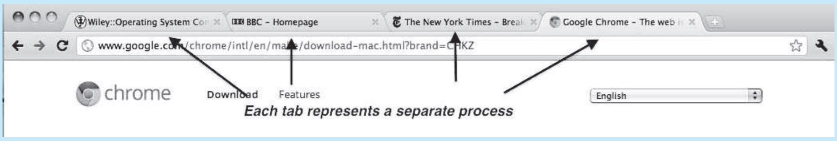
There are several reasons for providing an environment that allows process cooperation:

- **Information sharing**. Since several applications may be interested in the same piece of information (for instance, copying and pasting), we must provide an environment to allow concurrent access to such information.
- **Computation speedup**. If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing cores.
- **Modularity**. We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads, as we discussed in Chapter 2.

Cooperating processes require an **interprocess communication** (**IPC**) mechanism that will allow them to exchange data— that is, send data to and receive data from each other. There are two fundamental models of interprocess communication: **shared memory** and **message passing**. In the shared-memory model, a region of memory that is shared by the cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region. In the message-passing model,

### *MULTIPROCESS ARCHITECTURE—CHROME BROWSER*

Many websites contain active content, such as JavaScript, Flash, and HTML5 to provide a rich and dynamic web-browsing experience. Unfortunately, these web applications may also contain software bugs, which can result in sluggish response times and can even cause the web browser to crash. This isn't a big problem in a web browser that displays content from only one website. But most contemporary web browsers provide tabbed browsing, which allows a single instance of a web browser application to open several websites at the same time, with each site in a separate tab. To switch between the different sites, a user need only click on the appropriate tab. This arrangement is illustrated below:


Wiley::Operating System Co
BBC – Homepage
The New York Times – Breal
Google Chrome – The web
www.google.com/chrome/intl/en/mac/download-mac.html?brand=CHKZ
chrome
Download
Features
English

*Each tab represents a separate process*

A problem with this approach is that if a web application in any tab crashes, the entire process—including all other tabs displaying additional websites—crashes as well.

Google's Chrome web browser was designed to address this issue by using a multiprocess architecture. Chrome identifies three different types of processes: browser, renderers, and plug-ins.

- The **browser** process is responsible for managing the user interface as well as disk and network I/O. A new browser process is created when Chrome is started. Only one browser process is created.
- **Renderer** processes contain logic for rendering web pages. Thus, they contain the logic for handling HTML, Javascript, images, and so forth. As a general rule, a new renderer process is created for each website opened in a new tab, and so several renderer processes may be active at the same time.
- A **plug-in** process is created for each type of plug-in (such as Flash or QuickTime) in use. Plug-in processes contain the code for the plug-in as well as additional code that enables the plug-in to communicate with associated renderer processes and the browser process.

The advantage of the multiprocess approach is that websites run in isolation from one another. If one website crashes, only its renderer process is affected; all other processes remain unharmed. Furthermore, renderer processes run in a **sandbox**, which means that access to disk and network I/O is restricted, minimizing the effects of any security exploits.

communication takes place by means of messages exchanged between the cooperating processes. The two communications models are contrasted in Figure 3.11.

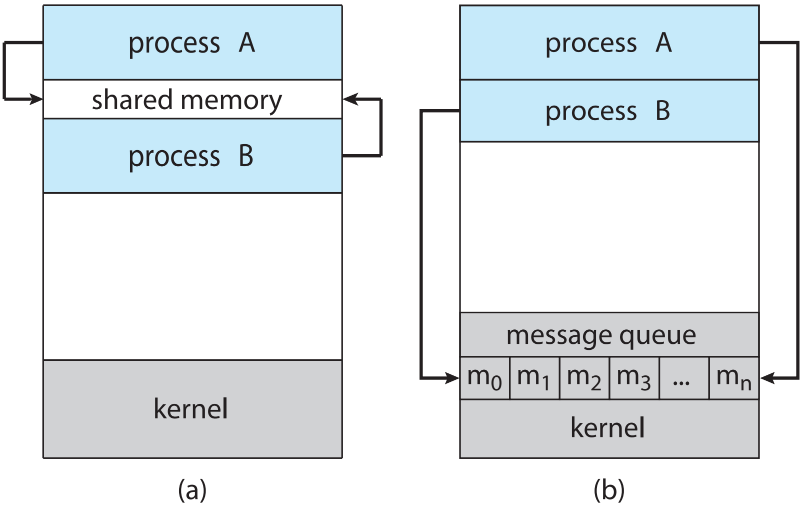


**Figure 3.11** Communications models. (a) Shared memory. (b) Message passing.

Both of the models just mentioned are common in operating systems, and many systems implement both. Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. Message passing is also easier to implement in a distributed system than shared memory. (Although there are systems that provide distributed shared memory, we do not consider them in this text.) Shared memory can be faster than message passing, since message-passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention. In shared-memory systems, system calls are required only to establish shared-memory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.

In Section 3.5 and Section 3.6 we explore shared-memory and message-passing systems in more detail.

## 3.5 IPC in Shared-Memory Systems

Interprocess communication using shared memory requires communicating processes to establish a region of shared memory. Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment. Other processes that wish to communicate using this shared-memory segment must attach it to their address space. Recall that, normally, the operating system tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas. The form of the data and the location are determined by these processes and are not under the operating system's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

To illustrate the concept of cooperating processes, let's consider the producer–consumer problem, which is a common paradigm for cooperating processes. A **producer** process produces information that is consumed by a **consumer** process. For example, a compiler may produce assembly code that is consumed by an assembler. The assembler, in turn, may produce object modules that are consumed by the loader. The producer–consumer problem also provides a useful metaphor for the client–server paradigm. We generally think of a server as a producer and a client as a consumer. For example, a web server produces (that is, provides) web content such as HTML files and images, which are consumed (that is, read) by the client web browser requesting the resource.

One solution to the producer–consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

Two types of buffers can be used. The **unbounded buffer** places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items. The **bounded buffer** assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

Let's look more closely at how the bounded buffer illustrates interprocess communication using shared memory. The following variables reside in a region of memory shared by the producer and consumer processes:

```
#define BUFFER_SIZE 10

typedef struct {
   . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

The shared `buffer` is implemented as a circular array with two logical pointers: `in` and `out`. The variable `in` points to the next free position in the buffer; `out` points to the first full position in the buffer. The buffer is empty when `in` == `out`; the buffer is full when ((`in` + 1) `%` `BUFFER_SIZE`) == `out`.

The code for the producer process is shown in Figure 3.12, and the code for the consumer process is shown in Figure 3.13. The producer process has a local variable `next_produced` in which the new item to be produced is stored. The consumer process has a local variable `next_consumed` in which the item to be consumed is stored.

This scheme allows at most `BUFFER_SIZE – 1` items in the buffer at the same time. We leave it as an exercise for you to provide a solution in which `BUFFER_SIZE` items can be in the buffer at the same time. In Section 3.7.1, we illustrate the POSIX API for shared memory.

```
item next_produced;

while (true) {
     /* produce an item in next_produced */

     while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
}
```

**Figure 3.12** The producer process using shared memory.

One issue this illustration does not address concerns the situation in which both the producer process and the consumer process attempt to access the shared buffer concurrently. In Chapter 6 and Chapter 7, we discuss how synchronization among cooperating processes can be implemented effectively in a shared-memory environment.

## 3.6 IPC in Message-Passing Systems

In Section 3.5, we showed how cooperating processes can communicate in a shared-memory environment. The scheme requires that these processes share a region of memory and that the code for accessing and manipulating the shared memory be written explicitly by the application programmer. Another way to achieve the same effect is for the operating system to provide the means for cooperating processes to communicate with each other via a message-passing facility.

```
item next_consumed;

while (true) {
     while (in == out)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;

     /* consume the item in next_consumed */
}
```

**Figure 3.13** The consumer process using shared memory.

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space. It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network. For example, an Internet `chat` program could be designed so that chat participants communicate with one another by exchanging messages.

A message-passing facility provides at least two operations:

```
send(message)
```

and

```
receive(message)
```

Messages sent by a process can be either fixed or variable in size. If only fixed-sized messages can be sent, the system-level implementation is straightforward. This restriction, however, makes the task of programming more difficult. Conversely, variable-sized messages require a more complex system-level implementation, but the programming task becomes simpler. This is a common kind of tradeoff seen throughout operating-system design.

If processes *P* and *Q* want to communicate, they must send messages to and receive messages from each other: a ***communication link*** must exist between them. This link can be implemented in a variety of ways. We are concerned here not with the link's physical implementation (such as shared memory, hardware bus, or network, which are covered in Chapter 19) but rather with its logical implementation. Here are several methods for logically implementing a link and the `send()`/`receive()` operations:

- Direct or indirect communication
- Synchronous or asynchronous communication
- Automatic or explicit buffering

We look at issues related to each of these features next.

### 3.6.1 Naming

Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication.

Under **direct communication**, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the `send()` and `receive()` primitives are defined as:

- `send(P, message)`—Send a `message` to process `P`.
- `receive(Q, message)`—Receive a `message` from process `Q`.

A communication link in this scheme has the following properties:

- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.

- A link is associated with exactly two processes.
- Between each pair of processes, there exists exactly one link.

This scheme exhibits ***symmetry*** in addressing; that is, both the sender process and the receiver process must name the other to communicate. A variant of this scheme employs ***asymmetry*** in addressing. Here, only the sender names the recipient; the recipient is not required to name the sender. In this scheme, the `send()` and `receive()` primitives are defined as follows:

- `send(P, message)`—Send a `message` to process `P`.
- `receive(id, message)`—Receive a `message` from any process. The variable `id` is set to the name of the process with which communication has taken place.

The disadvantage in both of these schemes (symmetric and asymmetric) is the limited modularity of the resulting process definitions. Changing the identifier of a process may necessitate examining all other process definitions. All references to the old identifier must be found, so that they can be modified to the new identifier. In general, any such ***hard-coding*** techniques, where identifiers must be explicitly stated, are less desirable than techniques involving indirection, as described next.

With ***indirect communication***, the messages are sent to and received from ***mailboxes***, or ***ports***. A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification. For example, POSIX message queues use an integer value to identify a mailbox. A process can communicate with another process via a number of different mailboxes, but two processes can communicate only if they have a shared mailbox. The `send()` and `receive()` primitives are defined as follows:

- `send(A, message)`—Send a `message` to mailbox `A`.
- `receive(A, message)`—Receive a `message` from mailbox `A`.

In this scheme, a communication link has the following properties:

- A link is established between a pair of processes only if both members of the pair have a shared mailbox.
- A link may be associated with more than two processes.
- Between each pair of communicating processes, a number of different links may exist, with each link corresponding to one mailbox.

Now suppose that processes $P_1$, $P_2$, and $P_3$ all share mailbox $A$. Process $P_1$ sends a message to $A$, while both $P_2$ and $P_3$ execute a `receive()` from $A$. Which process will receive the message sent by $P_1$? The answer depends on which of the following methods we choose:

- Allow a link to be associated with two processes at most.

- Allow at most one process at a time to execute a `receive()` operation.
- Allow the system to select arbitrarily which process will receive the message (that is, either $P_2$ or $P_3$, but not both, will receive the message). The system may define an algorithm for selecting which process will receive the message (for example, ***round robin,*** where processes take turns receiving messages). The system may identify the receiver to the sender.

A mailbox may be owned either by a process or by the operating system. If the mailbox is owned by a process (that is, the mailbox is part of the address space of the process), then we distinguish between the owner (which can only receive messages through this mailbox) and the user (which can only send messages to the mailbox). Since each mailbox has a unique owner, there can be no confusion about which process should receive a message sent to this mailbox. When a process that owns a mailbox terminates, the mailbox disappears. Any process that subsequently sends a message to this mailbox must be notified that the mailbox no longer exists.

In contrast, a mailbox that is owned by the operating system has an existence of its own. It is independent and is not attached to any particular process. The operating system then must provide a mechanism that allows a process to do the following:

- Create a new mailbox.
- Send and receive messages through the mailbox.
- Delete a mailbox.

The process that creates a new mailbox is that mailbox's owner by default. Initially, the owner is the only process that can receive messages through this mailbox. However, the ownership and receiving privilege may be passed to other processes through appropriate system calls. Of course, this provision could result in multiple receivers for each mailbox.

### 3.6.2 Synchronization

Communication between processes takes place through calls to `send()` and `receive()` primitives. There are different design options for implementing each primitive. Message passing may be either **blocking** or **nonblocking**—also known as **synchronous** and **asynchronous**. (Throughout this text, you will encounter the concepts of synchronous and asynchronous behavior in relation to various operating-system algorithms.)

- **Blocking send**. The sending process is blocked until the message is received by the receiving process or by the mailbox.
- **Nonblocking send**. The sending process sends the message and resumes operation.
- **Blocking receive**. The receiver blocks until a message is available.
- **Nonblocking receive**. The receiver retrieves either a valid message or a null.

```
message next_produced;

while (true) {
     /* produce an item in next_produced */

     send(next_produced);
}
```

**Figure 3.14** The producer process using message passing.

Different combinations of `send()` and `receive()` are possible. When both `send()` and `receive()` are blocking, we have a **rendezvous** between the sender and the receiver. The solution to the producer–consumer problem becomes trivial when we use blocking `send()` and `receive()` statements. The producer merely invokes the blocking `send()` call and waits until the message is delivered to either the receiver or the mailbox. Likewise, when the consumer invokes `receive()`, it blocks until a message is available. This is illustrated in Figures 3.14 and 3.15.

### 3.6.3 Buffering

Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:

- **Zero capacity**. The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
- **Bounded capacity**. The queue has finite length $n$; thus, at most $n$ messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without

```
message next_consumed;

while (true) {
     receive(next_consumed);

     /* consume the item in next_consumed */
}
```

**Figure 3.15** The consumer process using message passing.

waiting. The link's capacity is finite, however. If the link is full, the sender must block until space is available in the queue.

- **Unbounded capacity**. The queue's length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

The zero-capacity case is sometimes referred to as a message system with no buffering. The other cases are referred to as systems with automatic buffering.

## 3.7 Examples of IPC Systems

In this section, we explore four different IPC systems. We first cover the POSIX API for shared memory and then discuss message passing in the Mach operating system. Next, we present Windows IPC, which interestingly uses shared memory as a mechanism for providing certain types of message passing. We conclude with pipes, one of the earliest IPC mechanisms on UNIX systems.

### 3.7.1 POSIX Shared Memory

Several IPC mechanisms are available for POSIX systems, including shared memory and message passing. Here, we explore the POSIX API for shared memory.

POSIX shared memory is organized using memory-mapped files, which associate the region of shared memory with a file. A process must first create a shared-memory object using the `shm_open()` system call, as follows:

```
fd = shm_open(name, O_CREAT | O_RDWR, 0666);
```

The first parameter specifies the name of the shared-memory object. Processes that wish to access this shared memory must refer to the object by this name. The subsequent parameters specify that the shared-memory object is to be created if it does not yet exist (`O_CREAT`) and that the object is open for reading and writing (`O_RDWR`). The last parameter establishes the file-access permissions of the shared-memory object. A successful call to `shm_open()` returns an integer file descriptor for the shared-memory object.

Once the object is established, the `ftruncate()` function is used to configure the size of the object in bytes. The call

```
ftruncate(fd, 4096);
```

sets the size of the object to 4,096 bytes.

Finally, the `mmap()` function establishes a memory-mapped file containing the shared-memory object. It also returns a pointer to the memory-mapped file that is used for accessing the shared-memory object.

The programs shown in Figure 3.16 and Figure 3.17 use the producer–consumer model in implementing shared memory. The producer establishes a shared-memory object and writes to shared memory, and the consumer reads from shared memory.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

#include <sys/mman.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int fd;
/* pointer to shared memory obect */
char *ptr;

    /* create the shared memory object */
    fd = shm_open(name,O_CREAT | O_RDWR,0666);

    /* configure the size of the shared memory object */
    ftruncate(fd, SIZE);

    /* memory map the shared memory object */
    ptr = (char *)
     mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

    /* write to the shared memory object */
    sprintf(ptr,"%s",message_0);
    ptr += strlen(message_0);
    sprintf(ptr,"%s",message_1);
    ptr += strlen(message_1);

    return 0;
}
```

**Figure 3.16** Producer process illustrating POSIX shared-memory API.

The producer, shown in Figure 3.16, creates a shared-memory object named `OS` and writes the infamous string `"Hello World!"` to shared memory. The program memory-maps a shared-memory object of the specified size and allows writing to the object. The flag `MAP_SHARED` specifies that changes to the shared-memory object will be visible to all processes sharing the object. Notice that we write to the shared-memory object by calling the `sprintf()` function and writing the formatted string to the pointer `ptr`. After each write, we must increment the pointer by the number of bytes written.

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

#include <sys/mman.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int fd;
/* pointer to shared memory obect */
char *ptr;

    /* open the shared memory object */
    fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = (char *)
     mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

    /* read from the shared memory object */
    printf("%s",(char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```

**Figure 3.17** Consumer process illustrating POSIX shared-memory API.

The consumer process, shown in Figure 3.17, reads and outputs the contents of the shared memory. The consumer also invokes the `shm_unlink()` function, which removes the shared-memory segment after the consumer has accessed it. We provide further exercises using the POSIX shared-memory API in the programming exercises at the end of this chapter. Additionally, we provide more detailed coverage of memory mapping in Section 13.5.

### 3.7.2 Mach Message Passing

As an example of message passing, we next consider the Mach operating system. Mach was especially designed for distributed systems, but was shown to be suitable for desktop and mobile systems as well, as evidenced by its inclusion in the macOS and iOS operating systems, as discussed in Chapter 2.

The Mach kernel supports the creation and destruction of multiple ***tasks***, which are similar to processes but have multiple threads of control and fewer associated resources. Most communication in Mach—including all intertask communication—is carried out by **messages**. Messages are sent to, and received from, mailboxes, which are called **ports** in Mach. Ports are finite in size and unidirectional; for two-way communication, a message is sent to one port, and a response is sent to a separate ***reply*** port. Each port may have multiple senders, but only one receiver. Mach uses ports to represent resources such as tasks, threads, memory, and processors, while message passing provides an object-oriented approach for interacting with these system resources and services. Message passing may occur between any two ports on the same host or on separate hosts on a distributed system.

Associated with each port is a collection of **port rights** that identify the capabilities necessary for a task to interact with the port. For example, for a task to receive a message from a port, it must have the capability `MACH_PORT_RIGHT_RECEIVE` for that port. The task that creates a port is that port's owner, and the owner is the only task that is allowed to receive messages from that port. A port's owner may also manipulate the capabilities for a port. This is most commonly done in establishing a reply port. For example, assume that task $T1$ owns port $P1$, and it sends a message to port $P2$, which is owned by task $T2$. If $T1$ expects to receive a reply from $T2$, it must grant $T2$ the right `MACH_PORT_RIGHT_SEND` for port $P1$. Ownership of port rights is at the task level, which means that all threads belonging to the same task share the same port rights. Thus, two threads belonging to the same task can easily communicate by exchanging messages through the per-thread port associated with each thread.

When a task is created, two special ports—the ***Task Self*** port and the ***Notify*** port—are also created. The kernel has receive rights to the Task Self port, which allows a task to send messages to the kernel. The kernel can send notification of event occurrences to a task's Notify port (to which, of course, the task has receive rights).

The `mach_port_allocate()` function call creates a new port and allocates space for its queue of messages. It also identifies the rights for the port. Each port right represents a ***name*** for that port, and a port can only be accessed via a right. Port names are simple integer values and behave much like UNIX file descriptors. The following example illustrates creating a port using this API:

```
mach_port_t port; // the name of the port right

mach_port_allocate(
    mach_task_self(), // a task referring to itself
    MACH_PORT_RIGHT_RECEIVE, // the right for this port
    &port); // the name of the port right
```

Each task also has access to a **bootstrap port**, which allows a task to register a port it has created with a system-wide **bootstrap server**. Once a port has been registered with the bootstrap server, other tasks can look up the port in this registry and obtain rights for sending messages to the port.

The queue associated with each port is finite in size and is initially empty. As messages are sent to the port, the messages are copied into the queue. All messages are delivered reliably and have the same priority. Mach guarantees that multiple messages from the same sender are queued in first-in, first-out (FIFO) order but does not guarantee an absolute ordering. For instance, messages from two senders may be queued in any order.

Mach messages contain the following two fields:

- A fixed-size message header containing metadata about the message, including the size of the message as well as source and destination ports. Commonly, the sending thread expects a reply, so the port name of the source is passed on to the receiving task, which can use it as a “return address” in sending a reply.
- A variable-sized body containing data.

Messages may be either ***simple*** or ***complex***. A simple message contains ordinary, unstructured user data that are not interpreted by the kernel. A complex message may contain pointers to memory locations containing data (known as “out-of-line” data) or may also be used for transferring port rights to another task. Out-of-line data pointers are especially useful when a message must pass large chunks of data. A simple message would require copying and packaging the data in the message; out-of-line data transmission requires only a pointer that refers to the memory location where the data are stored.

The function `mach_msg()` is the standard API for both sending and receiving messages. The value of one of the function’s parameters—either `MACH_SEND_MSG` or `MACH_RCV_MSG`—indicates if it is a send or receive operation. We now illustrate how it is used when a client task sends a simple message to a server task. Assume there are two ports—`client` and `server`—associated with the client and server tasks, respectively. The code in Figure 3.18 shows the client task constructing a header and sending a message to the server, as well as the server task receiving the message sent from the client.

The `mach_msg()` function call is invoked by user programs for performing message passing. `mach_msg()` then invokes the function `mach_msg_trap()`, which is a system call to the Mach kernel. Within the kernel, `mach_msg_trap()` next calls the function `mach_msg_overwrite_trap()`, which then handles the actual passing of the message.

```
#include<mach/mach.h>

struct message {
  mach_msg_header_t header;
  int data;
};

mach_port_t client;
mach_port_t server;

      /* Client Code */

struct message message;

// construct the header
message.header.msgh_size = sizeof(message);
message.header.msgh_remote_port = server;
message.header.msgh_local_port = client;

// send the message
mach_msg(&message.header, // message header
  MACH_SEND_MSG, // sending a message
  sizeof(message), // size of message sent
  0, // maximum size of received message - unnecessary
  MACH_PORT_NULL, // name of receive port - unnecessary
  MACH_MSG_TIMEOUT_NONE, // no time outs
  MACH_PORT_NULL // no notify port
);

      /* Server Code */

struct message message;

// receive the message
mach_msg(&message.header, // message header
  MACH_RCV_MSG, // sending a message
  0, // size of message sent
  sizeof(message), // maximum size of received message
  server, // name of receive port
  MACH_MSG_TIMEOUT_NONE, // no time outs
  MACH_PORT_NULL // no notify port
);
```

**Figure 3.18** Example program illustrating message passing in Mach.

The send and receive operations themselves are flexible. For instance, when a message is sent to a port, its queue may be full. If the queue is not full, the message is copied to the queue, and the sending task continues. If the port's queue is full, the sender has several options (specified via parameters to `mach_msg()`:

1. Wait indefinitely until there is room in the queue.
2. Wait at most $n$ milliseconds.
3. Do not wait at all but rather return immediately.
4. Temporarily cache a message. Here, a message is given to the operating system to keep, even though the queue to which that message is being sent is full. When the message can be put in the queue, a notification message is sent back to the sender. Only one message to a full queue can be pending at any time for a given sending thread.

The final option is meant for server tasks. After finishing a request, a server task may need to send a one-time reply to the task that requested the service, but it must also continue with other service requests, even if the reply port for a client is full.

The major problem with message systems has generally been poor performance caused by copying of messages from the sender's port to the receiver's port. The Mach message system attempts to avoid copy operations by using virtual-memory-management techniques (Chapter 10). Essentially, Mach maps the address space containing the sender's message into the receiver's address space. Therefore, the message itself is never actually copied, as both the sender and receiver access the same memory. This message-management technique provides a large performance boost but works only for intrasystem messages.

### 3.7.3 Windows

The Windows operating system is an example of modern design that employs modularity to increase functionality and decrease the time needed to implement new features. Windows provides support for multiple operating environments, or ***subsystems.*** Application programs communicate with these subsystems via a message-passing mechanism. Thus, application programs can be considered clients of a subsystem server.

The message-passing facility in Windows is called the **advanced local procedure call** (**ALPC**) facility. It is used for communication between two processes on the same machine. It is similar to the standard remote procedure call (RPC) mechanism that is widely used, but it is optimized for and specific to Windows. (Remote procedure calls are covered in detail in Section 3.8.2.) Like Mach, Windows uses a port object to establish and maintain a connection between two processes. Windows uses two types of ports: **connection ports** and **communication ports**.

Server processes publish connection-port objects that are visible to all processes. When a client wants services from a subsystem, it opens a handle to the server's connection-port object and sends a connection request to that port. The server then creates a channel and returns a handle to the client. The channel consists of a pair of private communication ports: one for client–server messages, the other for server–client messages. Additionally, communication channels support a callback mechanism that allows the client and server to accept requests when they would normally be expecting a reply.

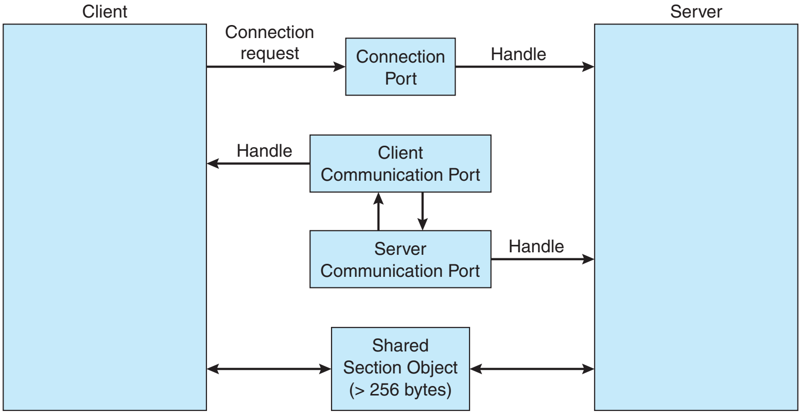

Client
Connection request
Connection Port
Handle
Server
Handle
Client Communication Port
Server Communication Port
Handle
Shared Section Object (> 256 bytes)

**Figure 3.19** Advanced local procedure calls in Windows.

When an ALPC channel is created, one of three message-passing techniques is chosen:

1. For small messages (up to 256 bytes), the port's message queue is used as intermediate storage, and the messages are copied from one process to the other.
2. Larger messages must be passed through a **section object**, which is a region of shared memory associated with the channel.
3. When the amount of data is too large to fit into a section object, an API is available that allows server processes to read and write directly into the address space of a client.

The client has to decide when it sets up the channel whether it will need to send a large message. If the client determines that it does want to send large messages, it asks for a section object to be created. Similarly, if the server decides that replies will be large, it creates a section object. So that the section object can be used, a small message is sent that contains a pointer and size information about the section object. This method is more complicated than the first method listed above, but it avoids data copying. The structure of advanced local procedure calls in Windows is shown in Figure 3.19.

It is important to note that the ALPC facility in Windows is not part of the Windows API and hence is not visible to the application programmer. Rather, applications using the Windows API invoke standard remote procedure calls. When the RPC is being invoked on a process on the same system, the RPC is handled indirectly through an ALPC procedure call. Additionally, many kernel services use ALPC to communicate with client processes.

### 3.7.4 Pipes

A **pipe** acts as a conduit allowing two processes to communicate. Pipes were one of the first IPC mechanisms in early UNIX systems. They typically provide one of the simpler ways for processes to communicate with one another, although they also have some limitations. In implementing a pipe, four issues must be considered:

1. Does the pipe allow bidirectional communication, or is communication unidirectional?
2. If two-way communication is allowed, is it half duplex (data can travel only one way at a time) or full duplex (data can travel in both directions at the same time)?
3. Must a relationship (such as ***parent–child***) exist between the communicating processes?
4. Can the pipes communicate over a network, or must the communicating processes reside on the same machine?

In the following sections, we explore two common types of pipes used on both UNIX and Windows systems: ordinary pipes and named pipes.

### 3.7.4.1 Ordinary Pipes

Ordinary pipes allow two processes to communicate in standard producer–consumer fashion: the producer writes to one end of the pipe (the **write end**) and the consumer reads from the other end (the **read end**). As a result, ordinary pipes are unidirectional, allowing only one-way communication. If two-way communication is required, two pipes must be used, with each pipe sending data in a different direction. We next illustrate constructing ordinary pipes on both UNIX and Windows systems. In both program examples, one process writes the message `Greetings` to the pipe, while the other process reads this message from the pipe.

On UNIX systems, ordinary pipes are constructed using the function

```
pipe(int fd[])
```

This function creates a pipe that is accessed through the `int fd[]` file descriptors: `fd[0]` is the read end of the pipe, and `fd[1]` is the write end. UNIX treats a pipe as a special type of file. Thus, pipes can be accessed using ordinary `read()` and `write()` system calls.

An ordinary pipe cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it creates via `fork()`. Recall from Section 3.3.1 that a child process inherits open files from its parent. Since a pipe is a special type of file, the child inherits the pipe from its parent process. Figure 3.20 illustrates

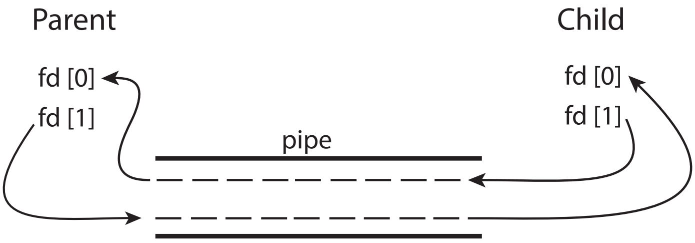


**Figure 3.20** File descriptors for an ordinary pipe.

```
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main(void)
{
   char write_msg[BUFFER_SIZE] = "Greetings";
   char read_msg[BUFFER_SIZE];
   int fd[2];
   pid_t pid;

      /* Program continues in  Figure 3.22 */
```

**Figure 3.21** Ordinary pipe in UNIX.

the relationship of the file descriptors in the `fd` array to the parent and child processes. As this illustrates, any writes by the parent to its write end of the pipe—`fd[1]`—can be read by the child from its read end—`fd[0]`—of the pipe.

In the UNIX program shown in Figure 3.21, the parent process creates a pipe and then sends a `fork()` call creating the child process. What occurs after the `fork()` call depends on how the data are to flow through the pipe. In this instance, the parent writes to the pipe, and the child reads from it. It is important to notice that both the parent process and the child process initially close their unused ends of the pipe. Although the program shown in Figure 3.21 does not require this action, it is an important step to ensure that a process reading from the pipe can detect end-of-file (`read()` returns 0) when the writer has closed its end of the pipe.

Ordinary pipes on Windows systems are termed **anonymous pipes**, and they behave similarly to their UNIX counterparts: they are unidirectional and employ parent–child relationships between the communicating processes. In addition, reading and writing to the pipe can be accomplished with the ordinary `ReadFile()` and `WriteFile()` functions. The Windows API for creating pipes is the `CreatePipe()` function, which is passed four parameters. The parameters provide separate handles for (1) reading and (2) writing to the pipe, as well as (3) an instance of the STARTUPINFO structure, which is used to specify that the child process is to inherit the handles of the pipe. Furthermore, (4) the size of the pipe (in bytes) may be specified.

Figure 3.23 illustrates a parent process creating an anonymous pipe for communicating with its child. Unlike UNIX systems, in which a child process automatically inherits a pipe created by its parent, Windows requires the programmer to specify which attributes the child process will inherit. This is

```
    /* create the pipe */
    if (pipe(fd) == -1) {
       fprintf(stderr,"Pipe failed");
       return 1;
    }

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }

    if (pid > 0) { /* parent process */
       /* close the unused end of the pipe */
       close(fd[READ_END]);

       /* write to the pipe */
       write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

       /* close the write end of the pipe */
       close(fd[WRITE_END]);
    }
    else { /* child process */
       /* close the unused end of the pipe */
       close(fd[WRITE_END]);

       /* read from the pipe */
       read(fd[READ_END], read_msg, BUFFER_SIZE);
       printf("read %s",read_msg);

       /* close the read end of the pipe */
       close(fd[READ_END]);
    }

    return 0;
}
```

**Figure 3.22** Figure 3.21, continued.

accomplished by first initializing the SECURITY_ATTRIBUTES structure to allow handles to be inherited and then redirecting the child process's handles for standard input or standard output to the read or write handle of the pipe. Since the child will be reading from the pipe, the parent must redirect the child's standard input to the read handle of the pipe. Furthermore, as the pipes are half duplex, it is necessary to prohibit the child from inheriting the write end of the

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

#define BUFFER_SIZE 25

int main(VOID)
{
  HANDLE ReadHandle, WriteHandle;
  STARTUPINFO si;
  PROCESS_INFORMATION pi;
  char message[BUFFER_SIZE] = "Greetings";
  DWORD written;

    /* Program continues in  Figure 3.24 */
```

**Figure 3.23** Windows anonymous pipe — parent process.

pipe. The program to create the child process is similar to the program in Figure 3.10, except that the fifth parameter is set to TRUE, indicating that the child process is to inherit designated handles from its parent. Before writing to the pipe, the parent first closes its unused read end of the pipe. The child process that reads from the pipe is shown in Figure 3.25. Before reading from the pipe, this program obtains the read handle to the pipe by invoking `GetStdHandle()`.

Note that ordinary pipes require a parent–child relationship between the communicating processes on both UNIX and Windows systems. This means that these pipes can be used only for communication between processes on the same machine.

### 3.7.4.2 Named Pipes

Ordinary pipes provide a simple mechanism for allowing a pair of processes to communicate. However, ordinary pipes exist only while the processes are communicating with one another. On both UNIX and Windows systems, once the processes have finished communicating and have terminated, the ordinary pipe ceases to exist.

Named pipes provide a much more powerful communication tool. Communication can be bidirectional, and no parent–child relationship is required. Once a named pipe is established, several processes can use it for communication. In fact, in a typical scenario, a named pipe has several writers. Additionally, named pipes continue to exist after communicating processes have finished. Both UNIX and Windows systems support named pipes, although the details of implementation differ greatly. Next, we explore named pipes in each of these systems.

Named pipes are referred to as FIFOs in UNIX systems. Once created, they appear as typical files in the file system. A FIFO is created with the `mkfifo()` system call and manipulated with the ordinary `open()`, `read()`, `write()`, and `close()` system calls. It will continue to exist until it is explicitly deleted

```
/* set up security attributes allowing pipes to be inherited */
SECURITY_ATTRIBUTES sa = {sizeof(SECURITY_ATTRIBUTES),NULL,TRUE};
/* allocate memory */
ZeroMemory(&pi, sizeof(pi));

/* create the pipe */
if (!CreatePipe(&ReadHandle, &WriteHandle, &sa, 0)) {
   fprintf(stderr, "Create Pipe Failed");
   return 1;
}

/* establish the START_INFO structure for the child process */
GetStartupInfo(&si);
si.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);

/* redirect standard input to the read end of the pipe */
si.hStdInput = ReadHandle;
si.dwFlags = STARTF_USESTDHANDLES;

/* don't allow the child to inherit the write end of pipe */
SetHandleInformation(WriteHandle, HANDLE_FLAG_INHERIT, 0);

/* create the child process */
CreateProcess(NULL, "child.exe", NULL, NULL,
 TRUE, /* inherit handles */
 0, NULL, NULL, &si, &pi);

/* close the unused end of the pipe */
CloseHandle(ReadHandle);

/* the parent writes to the pipe */
if (!WriteFile(WriteHandle, message,BUFFER_SIZE,&written,NULL))
   fprintf(stderr, "Error writing to pipe.");

/* close the write end of the pipe */
CloseHandle(WriteHandle);

/* wait for the child to exit */
WaitForSingleObject(pi.hProcess, INFINITE);
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
return 0;
}
```

**Figure 3.24** Figure 3.23, continued.

```
#include <stdio.h>
#include <windows.h>

#define BUFFER_SIZE 25

int main(VOID)
{
HANDLE Readhandle;
CHAR buffer[BUFFER_SIZE];
DWORD read;

   /* get the read handle of the pipe */
   ReadHandle = GetStdHandle(STD_INPUT_HANDLE);

   /* the child reads from the pipe */
   if (ReadFile(ReadHandle, buffer, BUFFER_SIZE, &read, NULL))
     printf("child read %s",buffer);
   else
     fprintf(stderr, "Error reading from pipe");

   return 0;
}
```

**Figure 3.25** Windows anonymous pipes — child process.

from the file system. Although FIFOs allow bidirectional communication, only half-duplex transmission is permitted. If data must travel in both directions, two FIFOs are typically used. Additionally, the communicating processes must reside on the same machine. If intermachine communication is required, sockets (Section 3.8.1) must be used.

Named pipes on Windows systems provide a richer communication mechanism than their UNIX counterparts. Full-duplex communication is allowed, and the communicating processes may reside on either the same or different machines. Additionally, only byte-oriented data may be transmitted across a UNIX FIFO, whereas Windows systems allow either byte- or message-oriented data. Named pipes are created with the `CreateNamedPipe()` function, and a client can connect to a named pipe using `ConnectNamedPipe()`. Communication over the named pipe can be accomplished using the `ReadFile()` and `WriteFile()` functions.

## 3.8 Communication in Client–Server Systems

In Section 3.4, we described how processes can communicate using shared memory and message passing. These techniques can be used for communication in client–server systems (Section 1.10.3) as well. In this section, we explore two other strategies for communication in client–server systems: sockets and

### *PIPES IN PRACTICE*

Pipes are used quite often in the UNIX command-line environment for situations in which the output of one command serves as input to another. For example, the UNIX `ls` command produces a directory listing. For especially long directory listings, the output may scroll through several screens. The command `less` manages output by displaying only one screen of output at a time where the user may use certain keys to move forward or backward in the file. Setting up a pipe between the `ls` and `less` commands (which are running as individual processes) allows the output of `ls` to be delivered as the input to `less`, enabling the user to display a large directory listing a screen at a time. A pipe can be constructed on the command line using the `|` character. The complete command is

```
ls | less
```

In this scenario, the `ls` command serves as the producer, and its output is consumed by the `less` command.

Windows systems provide a `more` command for the DOS shell with functionality similar to that of its UNIX counterpart `less`. (UNIX systems also provide a `more` command, but in the tongue-in-cheek style common in UNIX, the `less` command in fact provides *more* functionality than `more`!) The DOS shell also uses the `|` character for establishing a pipe. The only difference is that to get a directory listing, DOS uses the `dir` command rather than `ls`, as shown below:

```
dir | more
```

remote procedure calls (RPCs). As we shall see in our coverage of RPCs, not only are they useful for client–server computing, but Android also uses remote procedures as a form of IPC between processes running on the same system.

### 3.8.1 Sockets

A **socket** is defined as an endpoint for communication. A pair of processes communicating over a network employs a pair of sockets—one for each process. A socket is identified by an IP address concatenated with a port number. In general, sockets use a client–server architecture. The server waits for incoming client requests by listening to a specified port. Once a request is received, the server accepts a connection from the client socket to complete the connection. Servers implementing specific services (such as SSH, FTP, and HTTP) listen to ***well-known*** ports (an SSH server listens to port 22; an FTP server listens to port 21; and a web, or HTTP, server listens to port 80). All ports below 1024 are considered ***well known*** and are used to implement standard services.

When a client process initiates a request for a connection, it is assigned a port by its host computer. This port has some arbitrary number greater than 1024. For example, if a client on host `X` with IP address 146.86.5.20 wishes to establish a connection with a web server (which is listening on port 80) at

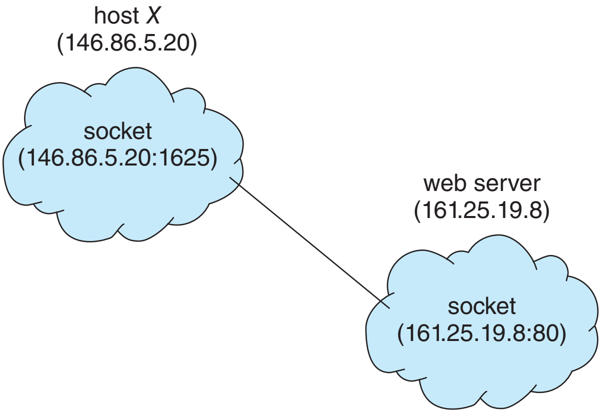


**Figure 3.26** Communication using sockets.

address 161.25.19.8, host `X` may be assigned port 1625. The connection will consist of a pair of sockets: (146.86.5.20:1625) on host `X` and (161.25.19.8:80) on the web server. This situation is illustrated in Figure 3.26. The packets traveling between the hosts are delivered to the appropriate process based on the destination port number.

All connections must be unique. Therefore, if another process also on host `X` wished to establish another connection with the same web server, it would be assigned a port number greater than 1024 and not equal to 1625. This ensures that all connections consist of a unique pair of sockets.

Although most program examples in this text use C, we will illustrate sockets using Java, as it provides a much easier interface to sockets and has a rich library for networking utilities. Those interested in socket programming in C or C++ should consult the bibliographical notes at the end of the chapter.

Java provides three different types of sockets. **Connection-oriented** (**TCP**) sockets are implemented with the `Socket` class. **Connectionless** (**UDP**) sockets use the `DatagramSocket` class. Finally, the `MulticastSocket` class is a subclass of the `DatagramSocket` class. A multicast socket allows data to be sent to multiple recipients.

Our example describes a date server that uses connection-oriented TCP sockets. The operation allows clients to request the current date and time from the server. The server listens to port 6013, although the port could have any arbitrary, unused number greater than 1024. When a connection is received, the server returns the date and time to the client.

The date server is shown in Figure 3.27. The server creates a `ServerSocket` that specifies that it will listen to port 6013. The server then begins listening to the port with the `accept()` method. The server blocks on the `accept()` method waiting for a client to request a connection. When a connection request is received, `accept()` returns a socket that the server can use to communicate with the client.

The details of how the server communicates with the socket are as follows. The server first establishes a `PrintWriter` object that it will use to communicate with the client. A `PrintWriter` object allows the server to write to the socket using the routine `print()` and `println()` methods for output. The

```
import java.net.*;
import java.io.*;

public class DateServer
{
  public static void main(String[] args) {
    try {
      ServerSocket sock = new ServerSocket(6013);

      /* now listen for connections */
      while (true) {
        Socket client = sock.accept();

        PrintWriter pout = new
         PrintWriter(client.getOutputStream(), true);

        /* write the Date to the socket */
        pout.println(new java.util.Date().toString());

        /* close the socket and resume */
        /* listening for connections */
        client.close();
      }
    }
    catch (IOException ioe) {
      System.err.println(ioe);
    }
  }
}
```

**Figure 3.27** Date server.

server process sends the date to the client, calling the method `println()`. Once it has written the date to the socket, the server closes the socket to the client and resumes listening for more requests.

A client communicates with the server by creating a socket and connecting to the port on which the server is listening. We implement such a client in the Java program shown in Figure 3.28. The client creates a `Socket` and requests a connection with the server at IP address 127.0.0.1 on port 6013. Once the connection is made, the client can read from the socket using normal stream I/O statements. After it has received the date from the server, the client closes the socket and exits. The IP address 127.0.0.1 is a special IP address known as the **loopback**. When a computer refers to IP address 127.0.0.1, it is referring to itself. This mechanism allows a client and server on the same host to communicate using the TCP/IP protocol. The IP address 127.0.0.1 could be replaced with the IP address of another host running the date server. In addition to an IP address, an actual host name, such as `www.westminstercollege.edu`, can be used as well.

```
import java.net.*;
import java.io.*;

public class DateClient
{
  public static void main(String[] args) {
    try {
      /* make connection to server socket */
      Socket sock = new Socket("127.0.0.1",6013);

      InputStream in = sock.getInputStream();
      BufferedReader bin = new
        BufferedReader(new InputStreamReader(in));

      /* read the date from the socket */
      String line;
      while ( (line = bin.readLine()) != null)
        System.out.println(line);

      /* close the socket connection*/
      sock.close();
    }
    catch (IOException ioe) {
      System.err.println(ioe);
    }
  }
}
```

**Figure 3.28** Date client.

Communication using sockets—although common and efficient—is considered a low-level form of communication between distributed processes. One reason is that sockets allow only an unstructured stream of bytes to be exchanged between the communicating threads. It is the responsibility of the client or server application to impose a structure on the data. In the next subsection, we look a higher-level method of communication: remote procedure calls (RPCs).

### 3.8.2 Remote Procedure Calls

One of the most common forms of remote service is the RPC paradigm, which was designed as a way to abstract the procedure-call mechanism for use between systems with network connections. It is similar in many respects to the IPC mechanism described in Section 3.4, and it is usually built on top of such a system. Here, however, because we are dealing with an environment in which the processes are executing on separate systems, we must use a message-based communication scheme to provide remote service.

In contrast to IPC messages, the messages exchanged in RPC communication are well structured and are thus no longer just packets of data. Each message is addressed to an RPC daemon listening to a port on the remote system, and each contains an identifier specifying the function to execute and the parameters to pass to that function. The function is then executed as requested, and any output is sent back to the requester in a separate message.

A **port** in this context is simply a number included at the start of a message packet. Whereas a system normally has one network address, it can have many ports within that address to differentiate the many network services it supports. If a remote process needs a service, it addresses a message to the proper port. For instance, if a system wished to allow other systems to be able to list its current users, it would have a daemon supporting such an RPC attached to a port—say, port 3027. Any remote system could obtain the needed information (that is, the list of current users) by sending an RPC message to port 3027 on the server. The data would be received in a reply message.

The semantics of RPCs allows a client to invoke a procedure on a remote host as it would invoke a procedure locally. The RPC system hides the details that allow communication to take place by providing a **stub** on the client side. Typically, a separate stub exists for each separate remote procedure. When the client invokes a remote procedure, the RPC system calls the appropriate stub, passing it the parameters provided to the remote procedure. This stub locates the port on the server and **marshals** the parameters. The stub then transmits a message to the server using message passing. A similar stub on the server side receives this message and invokes the procedure on the server. If necessary, return values are passed back to the client using the same technique. On Windows systems, stub code is compiled from a specification written in the **Microsoft Interface Definition Language** (**MIDL**), which is used for defining the interfaces between client and server programs.

Parameter marshaling addresses the issue concerning differences in data representation on the client and server machines. Consider the representation of 32-bit integers. Some systems (known as **big-endian**) store the most significant byte first, while other systems (known as **little-endian**) store the least significant byte first. Neither order is "better" per se; rather, the choice is arbitrary within a computer architecture. To resolve differences like this, many RPC systems define a machine-independent representation of data. One such representation is known as **external data representation** (**XDR**). On the client side, parameter marshaling involves converting the machine-dependent data into XDR before they are sent to the server. On the server side, the XDR data are unmarshaled and converted to the machine-dependent representation for the server.

Another important issue involves the semantics of a call. Whereas local procedure calls fail only under extreme circumstances, RPCs can fail, or be duplicated and executed more than once, as a result of common network errors. One way to address this problem is for the operating system to ensure that messages are acted on ***exactly once,*** rather than ***at most once.*** Most local procedure calls have the "exactly once" functionality, but it is more difficult to implement.

First, consider "at most once." This semantic can be implemented by attaching a timestamp to each message. The server must keep a history of all the timestamps of messages it has already processed or a history large enough to ensure that repeated messages are detected. Incoming messages that have a timestamp already in the history are ignored. The client can then send a message one or more times and be assured that it only executes once.

For "exactly once," we need to remove the risk that the server will never receive the request. To accomplish this, the server must implement the "at most once" protocol described above but must also acknowledge to the client that the RPC call was received and executed. These ACK messages are common throughout networking. The client must resend each RPC call periodically until it receives the ACK for that call.

Yet another important issue concerns the communication between a server and a client. With standard procedure calls, some form of binding takes place during link, load, or execution time (Chapter 9) so that a procedure call's name is replaced by the memory address of the procedure call. The RPC scheme requires a similar binding of the client and the server port, but how does a client know the port numbers on the server? Neither system has full information about the other, because they do not share memory.

Two approaches are common. First, the binding information may be predetermined, in the form of fixed port addresses. At compile time, an RPC call has a fixed port number associated with it. Once a program is compiled, the server cannot change the port number of the requested service. Second, binding can be done dynamically by a rendezvous mechanism. Typically, an operating system provides a rendezvous (also called a **matchmaker**) daemon on a fixed RPC port. A client then sends a message containing the name of the RPC to the rendezvous daemon requesting the port address of the RPC it needs to execute. The port number is returned, and the RPC calls can be sent to that port until the process terminates (or the server crashes). This method requires the extra overhead of the initial request but is more flexible than the first approach. Figure 3.29 shows a sample interaction.

The RPC scheme is useful in implementing a distributed file system (Chapter 19). Such a system can be implemented as a set of RPC daemons and clients. The messages are addressed to the distributed file system port on a server on which a file operation is to take place. The message contains the disk operation to be performed. The disk operation might be `read()`, `write()`, `rename()`, `delete()`, or `status()`, corresponding to the usual file-related system calls. The return message contains any data resulting from that call, which is executed by the DFS daemon on behalf of the client. For instance, a message might contain a request to transfer a whole file to a client or be limited to a simple block request. In the latter case, several requests may be needed if a whole file is to be transferred.

#### 3.8.2.1 Android RPC

Although RPCs are typically associated with client-server computing in a distributed system, they can also be used as a form of IPC between processes running on the same system. The Android operating system has a rich set of IPC mechanisms contained in its **binder** framework, including RPCs that allow one process to request services from another process.

Android defines an **application component** as a basic building block that provides utility to an Android application, and an app may combine multiple application components to provide functionality to an app. One such applica-

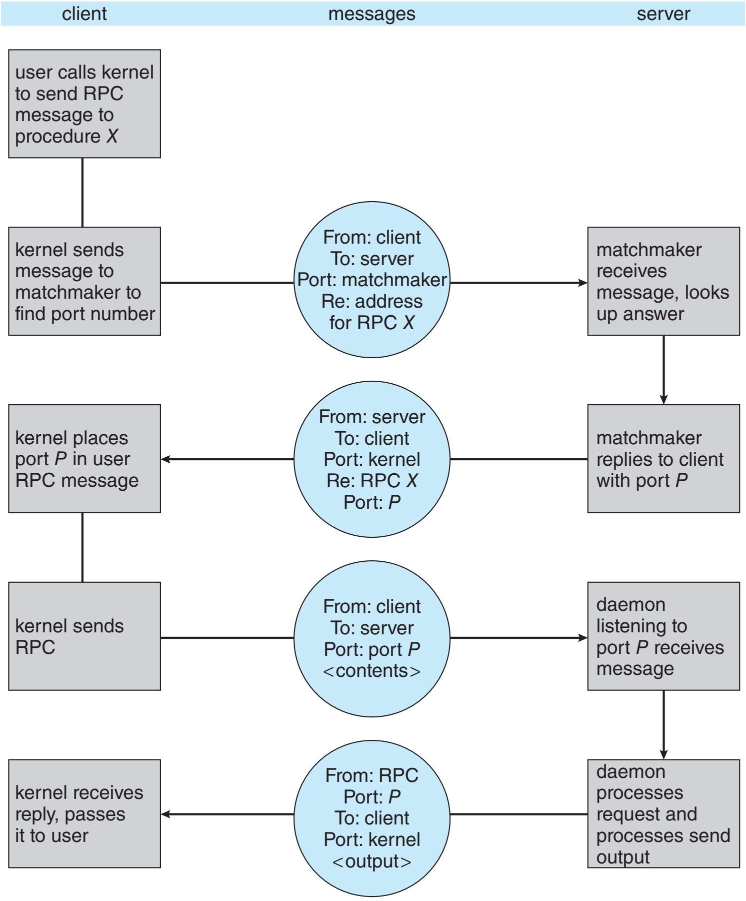

client
messages
server
user calls kernel to send RPC message to procedure *X*
kernel sends message to matchmaker to find port number
From: client To: server Port: matchmaker Re: address for RPC *X*
matchmaker receives message, looks up answer
matchmaker replies to client with port *P*
From: server To: client Port: kernel Re: RPC *X* Port: *P*
kernel places port *P* in user RPC message
kernel sends RPC
From: client To: server Port: port *P* <contents>
daemon listening to port *P* receives message
daemon processes request and processes send output
From: RPC Port: *P* To: client Port: kernel <output>
kernel receives reply, passes it to user

**Figure 3.29** Execution of a remote procedure call (RPC).

tion component is a **service**, which has no user interface but instead runs in the background while executing long-running operations or performing work for remote processes. Examples of services include playing music in the background and retrieving data over a network connection on behalf of another process, thereby preventing the other process from blocking as the data are being downloaded. When a client app invokes the `bindService()` method of a service, that service is "bound" and available to provide client-server communication using either message passing or RPCs.

A bound service must extend the Android class `Service` and must implement the method `onBind()`, which is invoked when a client calls `bindService()`. In the case of message passing, the `onBind()` method returns a `Messenger` service, which is used for sending messages from the client to the service. The `Messenger` service is only one-way; if the service must send a reply back to the client, the client must also provide a `Messenger` service, which is contained in the `replyTo` field of the `Message` object sent to the service. The service can then send messages back to the client.

To provide RPCs, the `onBind()` method must return an interface representing the methods in the remote object that clients use to interact with the service. This interface is written in regular Java syntax and uses the Android Interface Definition Language—AIDL—to create stub files, which serve as the client interface to remote services.

Here, we briefly outline the process required to provide a generic remote service named `remoteMethod()` using AIDL and the binder service. The interface for the remote service appears as follows:

```
/* RemoteService.aidl */
interface RemoteService
{
  boolean remoteMethod(int x, double y);
{
```

This file is written as `RemoteService.aidl`. The Android development kit will use it to generate a `.java` interface from the `.aidl` file, as well as a stub that serves as the RPC interface for this service. The server must implement the interface generated by the `.aidl` file, and the implementation of this interface will be called when the client invokes `remoteMethod()`.

When a client calls `bindService()`, the `onBind()` method is invoked on the server, and it returns the stub for the `RemoteService` object to the client. The client can then invoke the remote method as follows:

```
RemoteService service;
    . . .
service.remoteMethod(3, 0.14);
```

Internally, the Android binder framework handles parameter marshaling, transferring marshaled parameters between processes, and invoking the necessary implementation of the service, as well as sending any return values back to the client process.

## 3.9 Summary

- A process is a program in execution, and the status of the current activity of a process is represented by the program counter, as well as other registers.
- The layout of a process in memory is represented by four different sections: (1) text, (2) data, (3) heap, and (4) stack.
- As a process executes, it changes state. There are four general states of a process: (1) ready, (2) running, (3) waiting, and (4) terminated.
- A process control block (PCB) is the kernel data structure that represents a process in an operating system.
- The role of the process scheduler is to select an available process to run on a CPU.
- An operating system performs a context switch when it switches from running one process to running another.

- The `fork()` and `CreateProcess()` system calls are used to create processes on UNIX and Windows systems, respectively.
- When shared memory is used for communication between processes, two (or more) processes share the same region of memory. POSIX provides an API for shared memory.
- Two processes may communicate by exchanging messages with one another using message passing. The Mach operating system uses message passing as its primary form of interprocess communication. Windows provides a form of message passing as well.
- A pipe provides a conduit for two processes to communicate. There are two forms of pipes, ordinary and named. Ordinary pipes are designed for communication between processes that have a parent–child relationship. Named pipes are more general and allow several processes to communicate.
- UNIX systems provide ordinary pipes through the `pipe()` system call. Ordinary pipes have a read end and a write end. A parent process can, for example, send data to the pipe using its write end, and the child process can read it from its read end. Named pipes in UNIX are termed FIFOs.
- Windows systems also provide two forms of pipes—anonymous and named pipes. Anonymous pipes are similar to UNIX ordinary pipes. They are unidirectional and employ parent–child relationships between the communicating processes. Named pipes offer a richer form of interprocess communication than the UNIX counterpart, FIFOs.
- Two common forms of client–server communication are sockets and remote procedure calls (RPCs). Sockets allow two processes on different machines to communicate over a network. RPCs abstract the concept of function (procedure) calls in such a way that a function can be invoked on another process that may reside on a separate computer.
- The Android operating system uses RPCs as a form of interprocess communication using its binder framework.

## Practice Exercises

**3.1** Using the program shown in Figure 3.30, explain what the output will be at `LINE A`.

**3.2** Including the initial parent process, how many processes are created by the program shown in Figure 3.31?

**3.3** Original versions of Apple's mobile iOS operating system provided no means of concurrent processing. Discuss three major complications that concurrent processing adds to an operating system.

**3.4** Some computer systems provide multiple register sets. Describe what happens when a context switch occurs if the new context is already

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int value = 5;

int main()
{
pid_t pid;

  pid = fork();

  if (pid == 0) { /* child process */
    value += 15;
    return 0;
  }
  else if (pid > 0) { /* parent process */
    wait(NULL);
    printf("PARENT: value = %d",value); /* LINE A */
    return 0;
  }
}
```

**Figure 3.30** What output will be at Line A?

loaded into one of the register sets. What happens if the new context is in memory rather than in a register set and all the register sets are in use?

**3.5** When a process creates a new process using the `fork()` operation, which of the following states is shared between the parent process and the child process?

a. Stack

b. Heap

c. Shared memory segments

**3.6** Consider the "exactly once"semantic with respect to the RPC mechanism. Does the algorithm for implementing this semantic execute correctly even if the ACK message sent back to the client is lost due to a network problem? Describe the sequence of messages, and discuss whether "exactly once" is still preserved.

**3.7** Assume that a distributed system is susceptible to server failure. What mechanisms would be required to guarantee the "exactly once" semantic for execution of RPCs?

```
#include <stdio.h>
#include <unistd.h>

int main()
{
   /* fork a child process */
   fork();

   /* fork another child process */
   fork();

   /* and fork another */
   fork();

   return 0;
}
```

**Figure 3.31** How many processes are created?

## Further Reading

Process creation, management, and IPC in UNIX and Windows systems, respectively, are discussed in [Robbins and Robbins (2003)] and [Russinovich et al. (2017)]. [Love (2010)] covers support for processes in the Linux kernel, and [Hart (2005)] covers Windows systems programming in detail. Coverage of the multiprocess model used in Google's Chrome can be found at http://blog.chromium.org/2008/09/multi-process-architecture.html.

Message passing for multicore systems is discussed in [Holland and Seltzer (2011)]. [Levin (2013)] describes message passing in the Mach system, particularly with respect to macOS and iOS.

[Harold (2005)] provides coverage of socket programming in Java. Details on Android RPCs can be found at https://developer.android.com/guide/components/aidl.html. [Hart (2005)] and [Robbins and Robbins (2003)] cover pipes in Windows and UNIX systems, respectively.

Guidelines for Android development can be found at https://developer.android.com/guide/.

## Bibliography

**[Harold (2005)]** E. R. Harold, *Java Network Programming*, Third Edition, O'Reilly & Associates (2005).

**[Hart (2005)]** J. M. Hart, *Windows System Programming*, Third Edition, Addison-Wesley (2005).

**[Holland and Seltzer (2011)]** D. Holland and M. Seltzer, "Multicore OSes: Looking Forward from 1991, er, 2011", *Proceedings of the 13th USENIX conference on Hot topics in operating systems* (2011), pages 33–33.

**[Levin (2013)]** J. Levin, *Mac OS X and iOS Internals to the Apple's Core*, Wiley (2013).

**[Love (2010)]** R. Love, *Linux Kernel Development*, Third Edition, Developer's Library (2010).

**[Robbins and Robbins (2003)]** K. Robbins and S. Robbins, *Unix Systems Programming: Communication, Concurrency and Threads*, Second Edition, Prentice Hall (2003).

**[Russinovich et al. (2017)]** M. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals–Part 1,* Seventh Edition, Microsoft Press (2017).

## Chapter 3 Exercises

**3.8** Describe the actions taken by a kernel to context-switch between processes.

**3.9** Construct a process tree similar to Figure 3.7. To obtain process information for the UNIX or Linux system, use the command `ps -ael`. Use the command `man ps` to get more information about the `ps` command. The task manager on Windows systems does not provide the parent process ID, but the ***process monitor*** tool, available from `technet.microsoft.com`, provides a process-tree tool.

**3.10** Explain the role of the `init` (or `systemd`) process on UNIX and Linux systems in regard to process termination.

**3.11** Including the initial parent process, how many processes are created by the program shown in Figure 3.32?

**3.12** Explain the circumstances under which the line of code marked `printf("LINE J")` in Figure 3.33 will be reached.

**3.13** Using the program in Figure 3.34, identify the values of `pid` at lines `A`, `B`, `C`, and `D`. (Assume that the actual pids of the parent and child are 2600 and 2603, respectively.)

**3.14** Give an example of a situation in which ordinary pipes are more suitable than named pipes and an example of a situation in which named pipes are more suitable than ordinary pipes.

**3.15** Consider the RPC mechanism. Describe the undesirable consequences that could arise from not enforcing either the "at most once" or "exactly once" semantic. Describe possible uses for a mechanism that has neither of these guarantees.

**3.16** Using the program shown in Figure 3.35, explain what the output will be at lines `X` and `Y`.

```
#include <stdio.h>
#include <unistd.h>

int main()
{
   int i;

   for (i = 0; i < 4; i++)
      fork();

   return 0;
}
```

**Figure 3.21** How many processes are created?

**3.17** What are the benefits and the disadvantages of each of the following? Consider both the system level and the programmer level.

a. Synchronous and asynchronous communication

b. Automatic and explicit buffering

c. Send by copy and send by reference

d. Fixed-sized and variable-sized messages

---

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

   /* fork a child process */
   pid = fork();

   if (pid < 0) { /* error occurred */
     fprintf(stderr, "Fork Failed");
     return 1;
   }
   else if (pid == 0) { /* child process */
     execlp("/bin/ls","ls",NULL);
     printf("LINE J");
   }
   else { /* parent process */
     /* parent will wait for the child to complete */
     wait(NULL);
     printf("Child Complete");
   }

   return 0;
}
```

---

**Figure 3.22** When will `LINE J` be reached?

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid, pid1;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
    }
    else if (pid == 0) { /* child process */
      pid1 = getpid();
      printf("child: pid = %d",pid); /* A */
      printf("child: pid1 = %d",pid1); /* B */
    }
    else { /* parent process */
      pid1 = getpid();
      printf("parent: pid = %d",pid); /* C */
      printf("parent: pid1 = %d",pid1); /* D */
      wait(NULL);
    }

    return 0;
}
```

**Figure 3.23** What are the pid values?

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

#define SIZE 5

int nums[SIZE] = {0,1,2,3,4};

int main()
{
int i;
pid_t pid;

  pid = fork();

  if (pid == 0) {
    for (i = 0; i < SIZE; i++) {
      nums[i] *= -i;
      printf("CHILD: %d ",nums[i]); /* LINE X */
    }
  }
  else if (pid > 0) {
    wait(NULL);
    for (i = 0; i < SIZE; i++)
      printf("PARENT: %d ",nums[i]); /* LINE Y */
  }

  return 0;
}
```

**Figure 3.24** What output will be at Line X and Line Y?

## Programming Problems

**3.18** Using either a UNIX or a Linux system, write a C program that forks a child process that ultimately becomes a zombie process. This zombie process must remain in the system for at least 10 seconds. Process states can be obtained from the command

```
ps -l
```

The process states are shown below the `S` column; processes with a state of `Z` are zombies. The process identifier (pid) of the child process is listed in the `PID` column, and that of the parent is listed in the `PPID` column.

Perhaps the easiest way to determine that the child process is indeed a zombie is to run the program that you have written in the background (using the `&`) and then run the command `ps -l` to determine whether the child is a zombie process. Because you do not want too many zombie processes existing in the system, you will need to remove the one that you have created. The easiest way to do that is to terminate the parent process using the `kill` command. For example, if the pid of the parent is 4884, you would enter

```
kill -9 4884
```

**3.19** Write a C program called `time.c` that determines the amount of time necessary to run a command from the command line. This program will be run as "`./time <command>`" and will report the amount of elapsed time to run the specified command. This will involve using `fork()` and `exec()` functions, as well as the `gettimeofday()` function to determine the elapsed time. It will also require the use of two different IPC mechanisms.

The general strategy is to fork a child process that will execute the specified command. However, before the child executes the command, it will record a timestamp of the current time (which we term "starting time"). The parent process will wait for the child process to terminate. Once the child terminates, the parent will record the current timestamp for the ending time. The difference between the starting and ending times represents the elapsed time to execute the command. The example output below reports the amount of time to run the command `ls` :

```
./time ls
time.c
time

Elapsed time: 0.25422
```

As the parent and child are separate processes, they will need to arrange how the starting time will be shared between them. You will write two versions of this program, each representing a different method of IPC.

The first version will have the child process write the starting time to a region of shared memory before it calls `exec()`. After the child process terminates, the parent will read the starting time from shared memory. Refer to Section 3.7.1 for details using POSIX shared memory. In that section, there are separate programs for the producer and consumer. As the solution to this problem requires only a single program, the region of shared memory can be established before the child process is forked, allowing both the parent and child processes access to the region of shared memory.

The second version will use a pipe. The child will write the starting time to the pipe, and the parent will read from it following the termination of the child process.

You will use the `gettimeofday()` function to record the current timestamp. This function is passed a pointer to a `struct timeval` object, which contains two members: `tv_sec` and `t_usec`. These represent the number of elapsed seconds and microseconds since January 1, 1970 (known as the UNIX EPOCH). The following code sample illustrates how this function can be used:

```
struct timeval current;

gettimeofday(&current,NULL);

// current.tv_sec represents seconds
// current.tv_usec represents microseconds
```

For IPC between the child and parent processes, the contents of the shared memory pointer can be assigned the `struct timeval` representing the starting time. When pipes are used, a pointer to a `struct timeval` can be written to—and read from—the pipe.

**3.20** An operating system's **pid manager** is responsible for managing process identifiers. When a process is first created, it is assigned a unique pid by the pid manager. The pid is returned to the pid manager when the process completes execution, and the manager may later reassign this pid. Process identifiers are discussed more fully in Section 3.3.1. What is most important here is to recognize that process identifiers must be unique; no two active processes can have the same pid.

Use the following constants to identify the range of possible pid values:

```
#define MIN_PID 300
#define MAX_PID 5000
```

You may use any data structure of your choice to represent the availability of process identifiers. One strategy is to adopt what Linux has done and use a bitmap in which a value of 0 at position *i* indicates that a process id of value $i$ is available and a value of 1 indicates that the process id is currently in use.

Implement the following API for obtaining and releasing a pid:

- `int allocate_map(void)`—Creates and initializes a data structure for representing pids; returns −1 if unsuccessful, 1 if successful
- `int allocate_pid(void)`—Allocates and returns a pid; returns −1 if unable to allocate a pid (all pids are in use)
- `void release_pid(int pid)`—Releases a pid

This programming problem will be modified later on in Chapter 4 and in Chapter 6.

**3.21** The Collatz conjecture concerns what happens when we take any positive integer $n$ and apply the following algorithm:

$$n = \begin{cases} n/2, & \text{if n is even} \\ 3 \times n + 1, & \text{if n is odd} \end{cases}$$

The conjecture states that when this algorithm is continually applied, all positive integers will eventually reach 1. For example, if $n = 35$, the sequence is

$$35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1$$

Write a C program using the `fork()` system call that generates this sequence in the child process. The starting number will be provided from the command line. For example, if 8 is passed as a parameter on the command line, the child process will output $8, 4, 2, 1$. Because the parent and child processes have their own copies of the data, it will be necessary for the child to output the sequence. Have the parent invoke the `wait()` call to wait for the child process to complete before exiting the program. Perform necessary error checking to ensure that a positive integer is passed on the command line.

**3.22** In Exercise 3.21, the child process must output the sequence of numbers generated from the algorithm specified by the Collatz conjecture because the parent and child have their own copies of the data. Another approach to designing this program is to establish a shared-memory object between the parent and child processes. This technique allows the child to write the contents of the sequence to the shared-memory object. The parent can then output the sequence when the child completes. Because the memory is shared, any changes the child makes will be reflected in the parent process as well.

This program will be structured using POSIX shared memory as described in Section 3.7.1. The parent process will progress through the following steps:

a. Establish the shared-memory object (`shm_open()`, `ftruncate()`, and `mmap()`).

b. Create the child process and wait for it to terminate.

c. Output the contents of shared memory.

d. Remove the shared-memory object.

One area of concern with cooperating processes involves synchronization issues. In this exercise, the parent and child processes must be coordinated so that the parent does not output the sequence until the child finishes execution. These two processes will be synchronized using the `wait()` system call: the parent process will invoke `wait()`, which will suspend it until the child process exits.

**3.23** Section 3.8.1 describes certain port numbers as being well known—that is, they provide standard services. Port 17 is known as the *quote-of-the-day* service. When a client connects to port 17 on a server, the server responds with a quote for that day.

Modify the date server shown in Figure 3.27 so that it delivers a quote of the day rather than the current date. The quotes should be printable ASCII characters and should contain fewer than 512 characters, although multiple lines are allowed. Since these well-known ports are reserved and therefore unavailable, have your server listen to port 6017. The date client shown in Figure 3.28 can be used to read the quotes returned by your server.

**3.24** A **haiku** is a three-line poem in which the first line contains five syllables, the second line contains seven syllables, and the third line contains five syllables. Write a haiku server that listens to port 5575. When a client connects to this port, the server responds with a haiku. The date client shown in Figure 3.28 can be used to read the quotes returned by your haiku server.

**3.25** An echo server echoes back whatever it receives from a client. For example, if a client sends the server the string `Hello there!`, the server will respond with `Hello there!`

Write an echo server using the Java networking API described in Section 3.8.1. This server will wait for a client connection using the `accept()` method. When a client connection is received, the server will loop, performing the following steps:

- Read data from the socket into a buffer.
- Write the contents of the buffer back to the client.

The server will break out of the loop only when it has determined that the client has closed the connection.

The date server of Figure 3.27 uses the `java.io.BufferedReader` class. `BufferedReader` extends the `java.io.Reader` class, which is used for reading character streams. However, the echo server cannot guarantee that it will read characters from clients; it may receive binary data as well. The class `java.io.InputStream` deals with data at the byte level rather than the character level. Thus, your echo server must use an object that extends `java.io.InputStream`. The `read()` method in the `java.io.InputStream` class returns −1 when the client has closed its end of the socket connection.

**3.26** Design a program using ordinary pipes in which one process sends a string message to a second process, and the second process reverses the case of each character in the message and sends it back to the first process. For example, if the first process sends the message `Hi There`, the second process will return `hI tHERE`. This will require using two pipes, one for sending the original message from the first to the second process and the other for sending the modified message from the second to the first process. You can write this program using either UNIX or Windows pipes.

**3.27** Design a file-copying program named `filecopy.c` using ordinary pipes. This program will be passed two parameters: the name of the file to be copied and the name of the destination file. The program will then create an ordinary pipe and write the contents of the file to be copied to the pipe. The child process will read this file from the pipe and write it to the destination file. For example, if we invoke the program as follows:

```
./filecopy input.txt copy.txt
```

the file `input.txt` will be written to the pipe. The child process will read the contents of this file and write it to the destination file `copy.txt`. You may write this program using either UNIX or Windows pipes.

# Programming Projects

## Project 1—UNIX Shell

This project consists of designing a C program to serve as a shell interface that accepts user commands and then executes each command in a separate process. Your implementation will support input and output redirection, as well as pipes as a form of IPC between a pair of commands. Completing this project will involve using the UNIX `fork()`, `exec()`, `wait()`, `dup2()`, and `pipe()` system calls and can be completed on any Linux, UNIX, or macOS system.

### I. Overview

A shell interface gives the user a prompt, after which the next command is entered. The example below illustrates the prompt `osh>` and the user's next command: `cat prog.c`. (This command displays the file `prog.c` on the terminal using the UNIX `cat` command.)

```
osh>cat prog.c
```

One technique for implementing a shell interface is to have the parent process first read what the user enters on the command line (in this case, `cat prog.c`) and then create a separate child process that performs the command. Unless otherwise specified, the parent process waits for the child to exit before continuing. This is similar in functionality to the new process creation illustrated in Figure 3.9. However, UNIX shells typically also allow the child process to run in the background, or concurrently. To accomplish this, we add an ampersand (&) at the end of the command. Thus, if we rewrite the above command as

```
osh>cat prog.c &
```

the parent and child processes will run concurrently.

The separate child process is created using the `fork()` system call, and the user's command is executed using one of the system calls in the `exec()` family (as described in Section 3.3.1).

A C program that provides the general operations of a command-line shell is supplied in Figure 3.36. The `main()` function presents the prompt `osh->` and outlines the steps to be taken after input from the user has been read. The `main()` function continually loops as long as `should_run` equals 1; when the user enters `exit` at the prompt, your program will set `should_run` to 0 and terminate.

```
#include <stdio.h>
#include <unistd.h>

#define MAX_LINE 80 /* The maximum length command */

int main(void)
{
char *args[MAX_LINE/2 + 1]; /* command line arguments */
int should_run = 1; /* flag to determine when to exit program */

   while (should_run) {
      printf("osh>");
      fflush(stdout);

      /**
       * After reading user input, the steps are:
       * (1) fork a child process using fork()
       * (2) the child process will invoke execvp()
       * (3) parent will invoke wait() unless command included &
       */
   }

    return 0;
}
```

**Figure 3.36** Outline of simple shell.

This project is organized into several parts:

1. Creating the child process and executing the command in the child
2. Providing a history feature
3. Adding support of input and output redirection
4. Allowing the parent and child processes to communicate via a pipe

## II. Executing Command in a Child Process

The first task is to modify the `main()` function in Figure 3.36 so that a child process is forked and executes the command specified by the user. This will require parsing what the user has entered into separate tokens and storing the tokens in an array of character strings (`args` in Figure 3.36). For example, if the user enters the command `ps -ael` at the `osh>` prompt, the values stored in the `args` array are:

```
args[0] = "ps"
args[1] = "-ael"
args[2] = NULL
```

This `args` array will be passed to the `execvp()` function, which has the following prototype:

```
execvp(char *command, char *params[])
```

Here, `command` represents the command to be performed and `params` stores the parameters to this command. For this project, the `execvp()` function should be invoked as `execvp(args[0], args)`. Be sure to check whether the user included & to determine whether or not the parent process is to wait for the child to exit.

## III. Creating a History Feature

The next task is to modify the shell interface program so that it provides a ***history*** feature to allow a user to execute the most recent command by entering `!!`. For example, if a user enters the command `ls -l`, she can then execute that command again by entering `!!` at the prompt. Any command executed in this fashion should be echoed on the user's screen, and the command should also be placed in the history buffer as the next command.

Your program should also manage basic error handling. If there is no recent command in the history, entering `!!` should result in a message "`No commands in history.`"

## IV. Redirecting Input and Output

Your shell should then be modified to support the '>' and '<' redirection operators, where '>' redirects the output of a command to a file and '<' redirects the input to a command from a file. For example, if a user enters

```
osh>ls > out.txt
```

the output from the `ls` command will be redirected to the file `out.txt`. Similarly, input can be redirected as well. For example, if the user enters

```
osh>sort < in.txt
```

the file `in.txt` will serve as input to the `sort` command.

Managing the redirection of both input and output will involve using the `dup2()` function, which duplicates an existing file descriptor to another file descriptor. For example, if `fd` is a file descriptor to the file `out.txt`, the call

```
dup2(fd, STDOUT_FILENO);
```

duplicates `fd` to standard output (the terminal). This means that any writes to standard output will in fact be sent to the `out.txt` file.

You can assume that commands will contain either one input or one output redirection and will not contain both. In other words, you do not have to be concerned with command sequences such as `sort < in.txt > out.txt`.

### V. Communication via a Pipe

The final modification to your shell is to allow the output of one command to serve as input to another using a pipe. For example, the following command sequence

```
osh>ls -l | less
```

has the output of the command `ls -l` serve as the input to the `less` command. Both the `ls` and `less` commands will run as separate processes and will communicate using the UNIX `pipe()` function described in Section 3.7.4. Perhaps the easiest way to create these separate processes is to have the parent process create the child process (which will execute `ls -l`). This child will also create another child process (which will execute `less`) and will establish a pipe between itself and the child process it creates. Implementing pipe functionality will also require using the `dup2()` function as described in the previous section. Finally, although several commands can be chained together using multiple pipes, you can assume that commands will contain only one pipe character and will not be combined with any redirection operators.

## Project 2 —Linux Kernel Module for Task Information

In this project, you will write a Linux kernel module that uses the `/proc` file system for displaying a task's information based on its process identifier value `pid`. Before beginning this project, be sure you have completed the Linux kernel module programming project in Chapter 2, which involves creating an entry in the `/proc` file system. This project will involve writing a process identifier to the file `/proc/pid`. Once a `pid` has been written to the `/proc` file, subsequent reads from `/proc/pid` will report (1) the command the task is running, (2) the value of the task's `pid`, and (3) the current state of the task. An example of how your kernel module will be accessed once loaded into the system is as follows:

```
echo "1395" > /proc/pid
cat /proc/pid
command = [bash] pid = [1395] state = [1]
```

The `echo` command writes the characters `"1395"` to the `/proc/pid` file. Your kernel module will read this value and store its integer equivalent as it represents a process identifier. The `cat` command reads from `/proc/pid`, where your kernel module will retrieve the three fields from the `task_struct` associated with the task whose `pid` value is `1395`.

```
ssize_t proc_write(struct file *file, char __user *usr_buf,
  size_t count, loff_t *pos)
{
  int rv = 0;
  char *k_mem;

  /* allocate kernel memory */
  k_mem = kmalloc(count, GFP_KERNEL);

  /* copies user space usr_buf to kernel memory */
  copy_from_user(k_mem, usr_buf, count);

  printk(KERN_INFO "%s\n", k_mem);

  /* return kernel memory */
  kfree(k_mem);

  return count;
}
```

**Figure 3.37** The `proc_write()` function.

## I. Writing to the `/proc` File System

In the kernel module project in Chapter 2, you learned how to read from the `/proc` file system. We now cover how to write to `/proc`. Setting the field `.write` in `struct file_operations` to

```
.write = proc_write
```

causes the `proc_write()` function of Figure 3.37 to be called when a write operation is made to `/proc/pid`

The `kmalloc()` function is the kernel equivalent of the user-level `malloc()` function for allocating memory, except that kernel memory is being allocated. The `GFP_KERNEL` flag indicates routine kernel memory allocation. The `copy_from_user()` function copies the contents of `usr_buf` (which contains what has been written to `/proc/pid`) to the recently allocated kernel memory. Your kernel module will have to obtain the integer equivalent of this value using the kernel function `kstrtol()`, which has the signature

```
int kstrtol(const char *str, unsigned int base, long *res)
```

This stores the character equivalent of `str`, which is expressed as a `base` into `res`.

Finally, note that we return memory that was previously allocated with `kmalloc()` back to the kernel with the call to `kfree()`. Careful memory management—which includes releasing memory to prevent ***memory leaks***—is crucial when developing kernel-level code.

### II. Reading from the `/proc` File System

Once the process identifier has been stored, any reads from `/proc/pid` will return the name of the command, its process identifier, and its state. As illustrated in Section 3.1, the PCB in Linux is represented by the structure `task_struct`, which is found in the `<linux/sched.h>` include file. Given a process identifier, the function `pid_task()` returns the associated `task_struct`. The signature of this function appears as follows:

```
struct task_struct pid_task(struct pid *pid,
   enum pid_type type)
```

The kernel function `find_vpid(int pid)` can be used to obtain the `struct pid`, and `PIDTYPE_PID` can be used as the `pid_type`.

For a valid `pid` in the system, `pid_task` will return its `task_struct`. You can then display the values of the command, pid, and state. (You will probably have to read through the `task_struct` structure in `<linux/sched.h>` to obtain the names of these fields.)

If `pid_task()` is not passed a valid `pid`, it returns `NULL`. Be sure to perform appropriate error checking to check for this condition. If this situation occurs, the kernel module function associated with reading from `/proc/pid` should return 0.

In the source code download, we give the C program `pid.c`, which provides some of the basic building blocks for beginning this project.

## Project 3—Linux Kernel Module for Listing Tasks

In this project, you will write a kernel module that lists all current tasks in a Linux system. You will iterate through the tasks both linearly and depth first.

### Part I—Iterating over Tasks Linearly

In the Linux kernel, the `for_each_process()` macro easily allows iteration over all current tasks in the system:

```
#include <linux/sched.h>

struct task_struct *task;

for_each_process(task) {
   /* on each iteration task points to the next task */
}
```

The various fields in `task_struct` can then be displayed as the program loops through the `for_each_process()` macro.

### Assignment

Design a kernel module that iterates through all tasks in the system using the `for_each_process()` macro. In particular, output the task command, state, and process id of each task. (You will probably have to read through the `task_struct` structure in `<linux/sched.h>` to obtain the names of these fields.) Write this code in the module entry point so that its contents will appear in the kernel log buffer, which can be viewed using the `dmesg` command. To verify that your code is working correctly, compare the contents of the kernel log buffer with the output of the following command, which lists all tasks in the system:

```
ps -el
```

The two values should be very similar. Because tasks are dynamic, however, it is possible that a few tasks may appear in one listing but not the other.

## Part II—Iterating over Tasks with a Depth-First Search Tree

The second portion of this project involves iterating over all tasks in the system using a depth-first search (DFS) tree. (As an example: the DFS iteration of the processes in Figure 3.7 is 1, 8415, 8416, 9298, 9204, 2808, 3028, 3610, 4005.)

Linux maintains its process tree as a series of lists. Examining the `task_struct` in `<linux/sched.h>`, we see two `struct list_head` objects:

```
children
```

and

```
sibling
```

These objects are pointers to a list of the task's children, as well as its siblings. Linux also maintains a reference to the initial task in the system — `init_task` — which is of type `task_struct`. Using this information as well as macro operations on lists, we can iterate over the children of `init_task` as follows:

```
struct task_struct *task;
struct list_head *list;
```

```
list_for_each(list, &init_task->children) {
   task = list_entry(list, struct task_struct, sibling);
   /* task points to the next child in the list */
}
```

The `list_for_each()` macro is passed two parameters, both of type `struct list_head`:

- A pointer to the head of the list to be traversed
- A pointer to the head node of the list to be traversed

At each iteration of `list_for_each()`, the first parameter is set to the `list` structure of the next child. We then use this value to obtain each structure in the list using the `list_entry()` macro.

### Assignment

Beginning from `init_task` task, design a kernel module that iterates over all tasks in the system using a DFS tree. Just as in the first part of this project, output the name, state, and pid of each task. Perform this iteration in the kernel entry module so that its output appears in the kernel log buffer.

If you output all tasks in the system, you may see many more tasks than appear with the `ps -ael` command. This is because some threads appear as children but do not show up as ordinary processes. Therefore, to check the output of the DFS tree, use the command

```
ps -eLf
```

This command lists all tasks—including threads—in the system. To verify that you have indeed performed an appropriate DFS iteration, you will have to examine the relationships among the various tasks output by the ps command.

## Project 4—Kernel Data Structures

In Section 1.9, we covered various data structures that are common in operating systems. The Linux kernel provides several of these structures. Here, we explore using the circular, doubly linked list that is available to kernel developers. Much of what we discuss is available in the Linux source code—in this instance, the include file `<linux/list.h>`—and we recommend that you examine this file as you proceed through the following steps.

Initially, you must define a `struct` containing the elements that are to be inserted in the linked list. The following C `struct` defines a color as a mixture of red, blue, and green:

```
struct color {
   int red;
   int blue;
   int green;
```

```
    struct list_head list;
};
```

Notice the member `struct list_head list`. The `list_head` structure is defined in the include file `<linux/types.h>`, and its intention is to embed the linked list within the nodes that comprise the list. This `list_head` structure is quite simple—it merely holds two members, `next` and `prev`, that point to the next and previous entries in the list. By embedding the linked list within the structure, Linux makes it possible to manage the data structure with a series of ***macro*** functions.

## I. Inserting Elements into the Linked List

We can declare a `list_head` object, which we use as a reference to the head of the list by using the `LIST_HEAD()` macro:

```
static LIST_HEAD(color_list);
```

This macro defines and initializes the variable `color_list`, which is of type `struct list_head`.

We create and initialize instances of `struct color` as follows:

```
struct color *violet;

violet = kmalloc(sizeof(*violet), GFP_KERNEL);
violet->red = 138;
violet->blue = 43;
violet->green = 226;
INIT_LIST_HEAD(&violet->list);
```

The `kmalloc()` function is the kernel equivalent of the user-level `malloc()` function for allocating memory, except that kernel memory is being allocated. The `GFP_KERNEL` flag indicates routine kernel memory allocation. The macro `INIT_LIST_HEAD()` initializes the `list` member in `struct color`. We can then add this instance to the end of the linked list using the `list_add_tail()` macro:

```
list_add_tail(&violet->list, &color_list);
```

## II. Traversing the Linked List

Traversing the list involves using the `list_for_each_entry()` macro, which accepts three parameters:

- A pointer to the structure being iterated over
- A pointer to the head of the list being iterated over
- The name of the variable containing the `list_head` structure

The following code illustrates this macro:

```
struct color *ptr;

list_for_each_entry(ptr, &color_list, list) {
  /* on each iteration ptr points */
  /* to the next struct color */
}
```

## III. Removing Elements from the Linked List

Removing elements from the list involves using the `list_del()` macro, which is passed a pointer to `struct list_head`:

```
list_del(struct list_head *element);
```

This removes `element` from the list while maintaining the structure of the remainder of the list.

Perhaps the simplest approach for removing all elements from a linked list is to remove each element as you traverse the list. The macro `list_for_each_entry_safe()` behaves much like `list_for_each_entry()` except that it is passed an additional argument that maintains the value of the `next` pointer of the item being deleted. (This is necessary for preserving the structure of the list.) The following code example illustrates this macro:

```
struct color *ptr, *next;

list_for_each_entry_safe(ptr,next,&color_list,list) {
  /* on each iteration ptr points */
  /* to the next struct color */
  list_del(&ptr->list);
  kfree(ptr);
}
```

Notice that after deleting each element, we return memory that was previously allocated with `kmalloc()` back to the kernel with the call to `kfree()`.

## Part I—Assignment

In the module entry point, create a linked list containing four `struct color` elements. Traverse the linked list and output its contents to the kernel log buffer. Invoke the `dmesg` command to ensure that the list is properly constructed once the kernel module has been loaded.

In the module exit point, delete the elements from the linked list and return the free memory back to the kernel. Again, invoke the `dmesg` command to check that the list has been removed once the kernel module has been unloaded.

## Part II—Parameter Passing

This portion of the project will involve passing a parameter to a kernel module. The module will use this parameter as an initial value and generate the Collatz sequence as described in Exercise 3.21.

### Passing a Parameter to a Kernel Module

Parameters may be passed to kernel modules when they are loaded. For example, if the name of the kernel module is `collatz`, we can pass the initial value of 15 to the kernel parameter `start` as follows:

```
sudo insmod collatz.ko start=15
```

Within the kernel module, we declare `start` as a parameter using the following code:

```
#include<linux/moduleparam.h>

static int start = 25;

module_param(start, int, 0);
```

The `module_param()` macro is used to establish variables as parameters to kernel modules. `module_param()` is provided three arguments: (1) the name of the parameter, (2) its type, and (3) file permissions. Since we are not using a file system for accessing the parameter, we are not concerned with permissions and use a default value of 0. Note that the name of the parameter used with the `insmod` command must match the name of the associated kernel parameter. Finally, if we do not provide a value to the module parameter during loading with `insmod`, the default value (which in this case is 25) is used.

## Part II—Assignment

Design a kernel module named `collatz` that is passed an initial value as a module parameter. Your module will then generate and store the sequence in a kernel linked list when the module is loaded. Once the sequence has been stored, your module will traverse the list and output its contents to the kernel log buffer. Use the `dmesg` command to ensure that the sequence is properly generated once the module has been loaded.

In the module exit point, delete the contents of the list and return the free memory back to the kernel. Again, use `dmesg` to check that the list has been removed once the kernel module has been unloaded.

# CHAPTER 4

# Threads & Concurrency

The process model introduced in Chapter 3 assumed that a process was an executing program with a single thread of control. Virtually all modern operating systems, however, provide features enabling a process to contain multiple threads of control. Identifying opportunities for parallelism through the use of threads is becoming increasingly important for modern multicore systems that provide multiple CPUs.

In this chapter, we introduce many concepts, as well as challenges, associated with multithreaded computer systems, including a discussion of the APIs for the Pthreads, Windows, and Java thread libraries. Additionally, we explore several new features that abstract the concept of creating threads, allowing developers to focus on identifying opportunities for parallelism and letting language features and API frameworks manage the details of thread creation and management. We look at a number of issues related to multithreaded programming and its effect on the design of operating systems. Finally, we explore how the Windows and Linux operating systems support threads at the kernel level.

## CHAPTER OBJECTIVES

- Identify the basic components of a thread, and contrast threads and processes.
- Describe the major benefits and significant challenges of designing multithreaded processes.
- Illustrate different approaches to implicit threading, including thread pools, fork-join, and Grand Central Dispatch.
- Describe how the Windows and Linux operating systems represent threads.
- Design multithreaded applications using the Pthreads, Java, and Windows threading APIs.

## 4.1 Overview

A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter (PC), a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals. A traditional process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time. Figure 4.1 illustrates the difference between a traditional **single-threaded** process and a **multithreaded** process.

### 4.1.1 Motivation

Most software applications that run on modern computers and mobile devices are multithreaded. An application typically is implemented as a separate process with several threads of control. Below we highlight a few examples of multithreaded applications:

- An application that creates photo thumbnails from a collection of images may use a separate thread to generate a thumbnail from each separate image.
- A web browser might have one thread display images or text while another thread retrieves data from the network.
- A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.

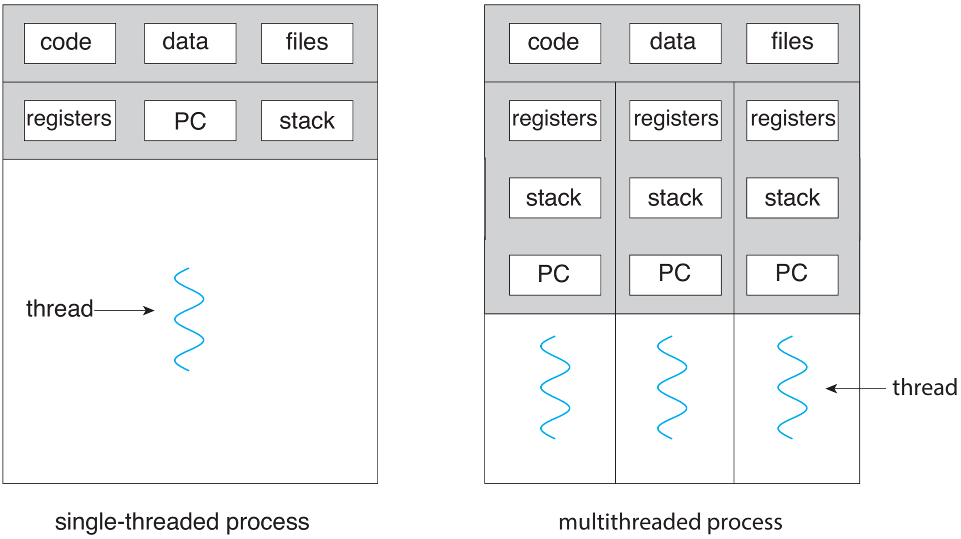

code
data
files
registers
PC
stack
thread
code
data
files
registers
registers
registers
stack
stack
stack
PC
PC
PC
thread
single-threaded process
multithreaded process

**Figure 4.1** Single-threaded and multithreaded processes.

Applications can also be designed to leverage processing capabilities on multicore systems. Such applications can perform several CPU-intensive tasks in parallel across the multiple computing cores.

In certain situations, a single application may be required to perform several similar tasks. For example, a web server accepts client requests for web pages, images, sound, and so forth. A busy web server may have several (perhaps thousands of) clients concurrently accessing it. If the web server ran as a traditional single-threaded process, it would be able to service only one client at a time, and a client might have to wait a very long time for its request to be serviced.

One solution is to have the server run as a single process that accepts requests. When the server receives a request, it creates a separate process to service that request. In fact, this process-creation method was in common use before threads became popular. Process creation is time consuming and resource intensive, however. If the new process will perform the same tasks as the existing process, why incur all that overhead? It is generally more efficient to use one process that contains multiple threads. If the web-server process is multithreaded, the server will create a separate thread that listens for client requests. When a request is made, rather than creating another process, the server creates a new thread to service the request and resumes listening for additional requests. This is illustrated in Figure 4.2.

Most operating system kernels are also typically multithreaded. As an example, during system boot time on Linux systems, several kernel threads are created. Each thread performs a specific task, such as managing devices, memory management, or interrupt handling. The command `ps -ef` can be used to display the kernel threads on a running Linux system. Examining the output of this command will show the kernel thread `kthreadd` (with `pid` = 2), which serves as the parent of all other kernel threads.

Many applications can also take advantage of multiple threads, including basic sorting, trees, and graph algorithms. In addition, programmers who must solve contemporary CPU-intensive problems in data mining, graphics, and artificial intelligence can leverage the power of modern multicore systems by designing solutions that run in parallel.

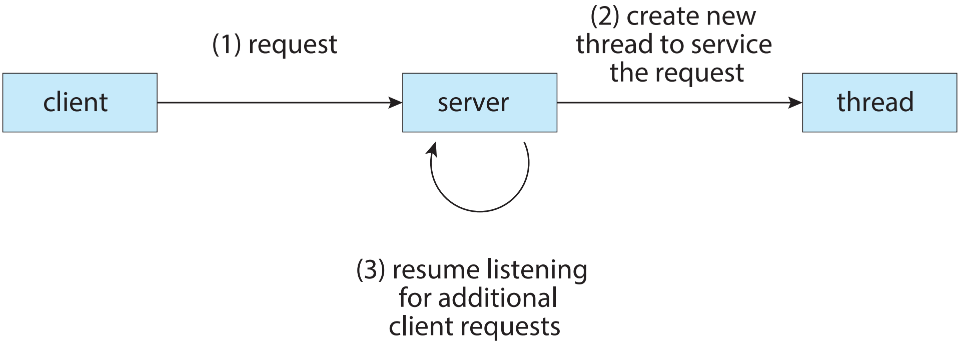

(1) request
(2) create new thread to service the request
client
server
thread
(3) resume listening for additional client requests

**Figure 4.2** Multithreaded server architecture.

### 4.1.2 Benefits

The benefits of multithreaded programming can be broken down into four major categories:

1. **Responsiveness**. Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. This quality is especially useful in designing user interfaces. For instance, consider what happens when a user clicks a button that results in the performance of a time-consuming operation. A single-threaded application would be unresponsive to the user until the operation had been completed. In contrast, if the time-consuming operation is performed in a separate, asynchronous thread, the application remains responsive to the user.
2. **Resource sharing**. Processes can share resources only through techniques such as shared memory and message passing. Such techniques must be explicitly arranged by the programmer. However, threads share the memory and the resources of the process to which they belong by default. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.
3. **Economy**. Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads. Empirically gauging the difference in overhead can be difficult, but in general thread creation consumes less time and memory than process creation. Additionally, context switching is typically faster between threads than between processes.
4. **Scalability.** The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores. A single-threaded process can run on only one processor, regardless how many are available. We explore this issue further in the following section.

## 4.2 Multicore Programming

Earlier in the history of computer design, in response to the need for more computing performance, single-CPU systems evolved into multi-CPU systems. A later, yet similar, trend in system design is to place multiple computing cores on a single processing chip where each core appears as a separate CPU to the operating system (Section 1.3.2). We refer to such systems as **multicore**, and multithreaded programming provides a mechanism for more efficient use of these multiple computing cores and improved concurrency. Consider an application with four threads. On a system with a single computing core, concurrency merely means that the execution of the threads will be interleaved over time (Figure 4.3), because the processing core is capable of executing only one thread at a time. On a system with multiple cores, however, concurrency

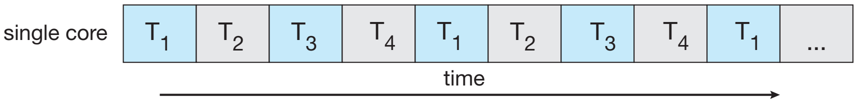


**Figure 4.3** Concurrent execution on a single-core system.

means that some threads can run in parallel, because the system can assign a separate thread to each core (Figure 4.4).

Notice the distinction between ***concurrency*** and ***parallelism*** in this discussion. A concurrent system supports more than one task by allowing all the tasks to make progress. In contrast, a parallel system can perform more than one task simultaneously. Thus, it is possible to have concurrency without parallelism. Before the advent of multiprocessor and multicore architectures, most computer systems had only a single processor, and CPU schedulers were designed to provide the illusion of parallelism by rapidly switching between processes, thereby allowing each process to make progress. Such processes were running concurrently, but not in parallel.

### 4.2.1 Programming Challenges

The trend toward multicore systems continues to place pressure on system designers and application programmers to make better use of the multiple computing cores. Designers of operating systems must write scheduling algorithms that use multiple processing cores to allow the parallel execution shown in Figure 4.4. For application programmers, the challenge is to modify existing programs as well as design new programs that are multithreaded.

In general, five areas present challenges in programming for multicore systems:

1. **Identifying tasks**. This involves examining applications to find areas that can be divided into separate, concurrent tasks. Ideally, tasks are independent of one another and thus can run in parallel on individual cores.
2. **Balance**. While identifying tasks that can run in parallel, programmers must also ensure that the tasks perform equal work of equal value. In some instances, a certain task may not contribute as much value to the overall process as other tasks. Using a separate execution core to run that task may not be worth the cost.

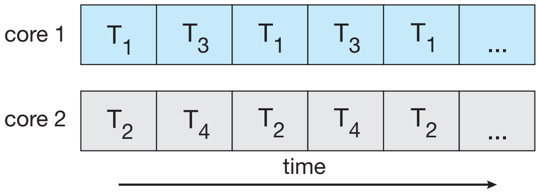


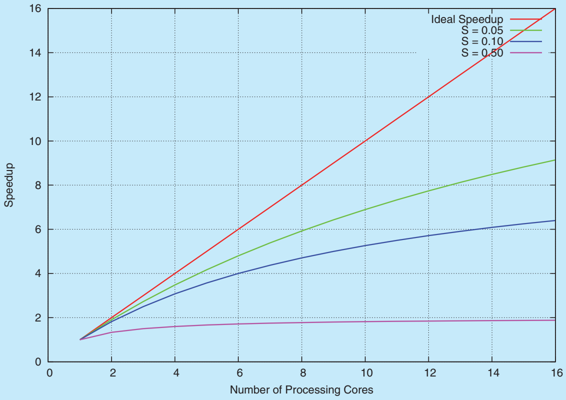
**Figure 4.4** Parallel execution on a multicore system.

### *AMDAHL'S LAW*

Amdahl's Law is a formula that identifies potential performance gains from adding additional computing cores to an application that has both serial (nonparallel) and parallel components. If $S$ is the portion of the application that must be performed serially on a system with $N$ processing cores, the formula appears as follows:

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

As an example, assume we have an application that is 75 percent parallel and 25 percent serial. If we run this application on a system with two processing cores, we can get a speedup of 1.6 times. If we add two additional cores (for a total of four), the speedup is 2.28 times. Below is a graph illustrating Amdahl's Law in several different scenarios.



One interesting fact about Amdahl's Law is that as $N$ approaches infinity, the speedup converges to $1/S$. For example, if 50 percent of an application is performed serially, the maximum speedup is 2.0 times, regardless of the number of processing cores we add. This is the fundamental principle behind Amdahl's Law: the serial portion of an application can have a disproportionate effect on the performance we gain by adding additional computing cores.

3. **Data splitting**. Just as applications are divided into separate tasks, the data accessed and manipulated by the tasks must be divided to run on separate cores.
4. **Data dependency**. The data accessed by the tasks must be examined for dependencies between two or more tasks. When one task depends on data from another, programmers must ensure that the execution of the tasks is synchronized to accommodate the data dependency. We examine such strategies in Chapter 6.

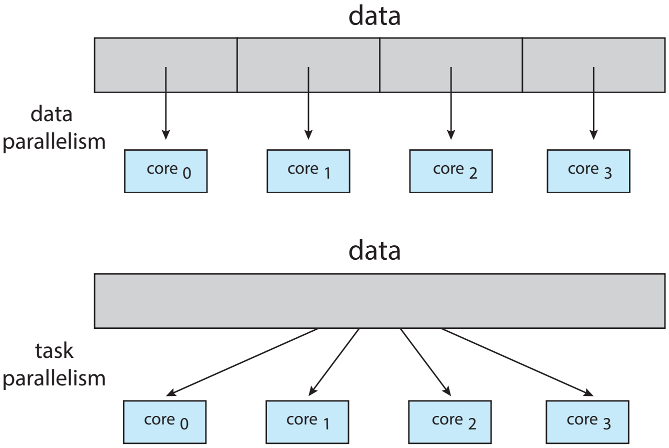


**Figure 4.5** Data and task parallelism.

5. **Testing and debugging**. When a program is running in parallel on multiple cores, many different execution paths are possible. Testing and debugging such concurrent programs is inherently more difficult than testing and debugging single-threaded applications.

Because of these challenges, many software developers argue that the advent of multicore systems will require an entirely new approach to designing software systems in the future. (Similarly, many computer science educators believe that software development must be taught with increased emphasis on parallel programming.)

### 4.2.2 Types of Parallelism

In general, there are two types of parallelism: data parallelism and task parallelism. **Data parallelism** focuses on distributing subsets of the same data across multiple computing cores and performing the same operation on each core. Consider, for example, summing the contents of an array of size $N$. On a single-core system, one thread would simply sum the elements $[0] \ldots [N-1]$. On a dual-core system, however, thread $A$, running on core 0, could sum the elements $[0] \ldots [N/2-1]$ while thread $B$, running on core 1, could sum the elements $[N/2] \ldots [N-1]$. The two threads would be running in parallel on separate computing cores.

**Task parallelism** involves distributing not data but tasks (threads) across multiple computing cores. Each thread is performing a unique operation. Different threads may be operating on the same data, or they may be operating on different data. Consider again our example above. In contrast to that situation, an example of task parallelism might involve two threads, each performing a unique statistical operation on the array of elements. The threads again are operating in parallel on separate computing cores, but each is performing a unique operation.

Fundamentally, then, data parallelism involves the distribution of data across multiple cores, and task parallelism involves the distribution of tasks across multiple cores, as shown in Figure 4.5. However, data and task paral-

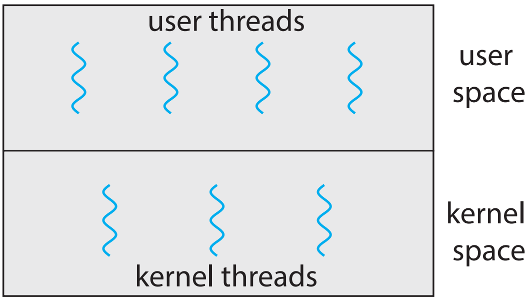


**Figure 4.6** User and kernel threads.

lelism are not mutually exclusive, and an application may in fact use a hybrid of these two strategies.

## 4.3 Multithreading Models

Our discussion so far has treated threads in a generic sense. However, support for threads may be provided either at the user level, for **user threads**, or by the kernel, for **kernel threads**. User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system. Virtually all contemporary operating systems—including Windows, Linux, and macOS— support kernel threads.

Ultimately, a relationship must exist between user threads and kernel threads, as illustrated in Figure 4.6. In this section, we look at three common ways of establishing such a relationship: the many-to-one model, the one-to-one model, and the many-to-many model.

### 4.3.1 Many-to-One Model

The many-to-one model (Figure 4.7) maps many user-level threads to one kernel thread. Thread management is done by the thread library in user space, so it is efficient (we discuss thread libraries in Section 4.4). However, the entire process will block if a thread makes a blocking system call. Also, because only

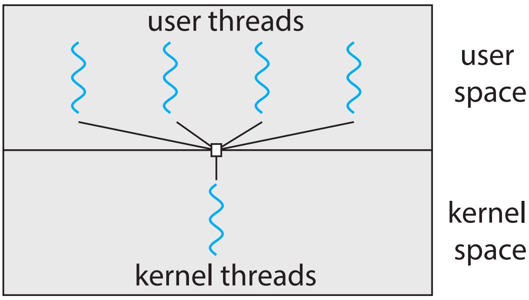


**Figure 4.7** Many-to-one model.

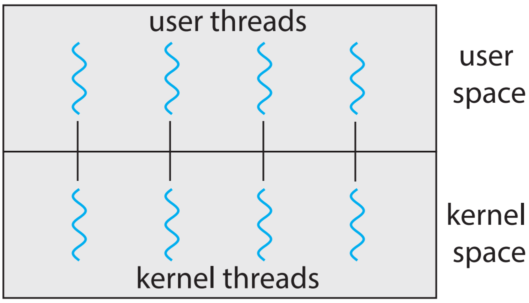


**Figure 4.8** One-to-one model.

one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems. **Green threads**—a thread library available for Solaris systems and adopted in early versions of Java—used the many-to-one model. However, very few systems continue to use the model because of its inability to take advantage of multiple processing cores, which have now become standard on most computer systems.

### 4.3.2 One-to-One Model

The one-to-one model (Figure 4.8) maps each user thread to a kernel thread. It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call. It also allows multiple threads to run in parallel on multiprocessors. The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread, and a large number of kernel threads may burden the performance of a system. Linux, along with the family of Windows operating systems, implement the one-to-one model.

### 4.3.3 Many-to-Many Model

The many-to-many model (Figure 4.9) multiplexes many user-level threads to a smaller or equal number of kernel threads. The number of kernel threads may be specific to either a particular application or a particular machine (an application may be allocated more kernel threads on a system with eight processing cores than a system with four cores).

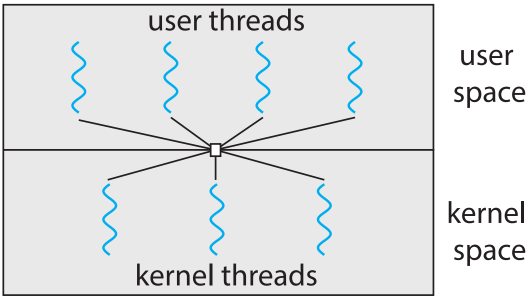


**Figure 4.9** Many-to-many model.

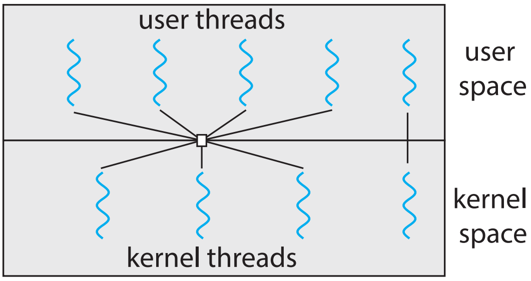


**Figure 4.10** Two-level model.

Let's consider the effect of this design on concurrency. Whereas the many-to-one model allows the developer to create as many user threads as she wishes, it does not result in parallelism, because the kernel can schedule only one kernel thread at a time. The one-to-one model allows greater concurrency, but the developer has to be careful not to create too many threads within an application. (In fact, on some systems, she may be limited in the number of threads she can create.) The many-to-many model suffers from neither of these shortcomings: developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.

One variation on the many-to-many model still multiplexes many user-level threads to a smaller or equal number of kernel threads but also allows a user-level thread to be bound to a kernel thread. This variation is sometimes referred to as the **two-level model** (Figure 4.10).

Although the many-to-many model appears to be the most flexible of the models discussed, in practice it is difficult to implement. In addition, with an increasing number of processing cores appearing on most systems, limiting the number of kernel threads has become less important. As a result, most operating systems now use the one-to-one model. However, as we shall see in Section 4.5, some contemporary concurrency libraries have developers identify tasks that are then mapped to threads using the many-to-many model.

## 4.4 Thread Libraries

A **thread library** provides the programmer with an API for creating and managing threads. There are two primary ways of implementing a thread library. The first approach is to provide a library entirely in user space with no kernel support. All code and data structures for the library exist in user space. This means that invoking a function in the library results in a local function call in user space and not a system call.

The second approach is to implement a kernel-level library supported directly by the operating system. In this case, code and data structures for the library exist in kernel space. Invoking a function in the API for the library typically results in a system call to the kernel.

Three main thread libraries are in use today: POSIX Pthreads, Windows, and Java. Pthreads, the threads extension of the POSIX standard, may be provided as either a user-level or a kernel-level library. The Windows thread library is a kernel-level library available on Windows systems. The Java thread API allows threads to be created and managed directly in Java programs. However, because in most instances the JVM is running on top of a host operating system, the Java thread API is generally implemented using a thread library available on the host system. This means that on Windows systems, Java threads are typically implemented using the Windows API; UNIX, Linux, and macOS systems typically use Pthreads.

For POSIX and Windows threading, any data declared globally—that is, declared outside of any function—are shared among all threads belonging to the same process. Because Java has no equivalent notion of global data, access to shared data must be explicitly arranged between threads.

In the remainder of this section, we describe basic thread creation using these three thread libraries. As an illustrative example, we design a multithreaded program that performs the summation of a non-negative integer in a separate thread using the well-known summation function:

$$sum = \sum_{i=1}^{N} i$$

For example, if $N$ were 5, this function would represent the summation of integers from 1 to 5, which is 15. Each of the three programs will be run with the upper bounds of the summation entered on the command line. Thus, if the user enters 8, the summation of the integer values from 1 to 8 will be output.

Before we proceed with our examples of thread creation, we introduce two general strategies for creating multiple threads: **asynchronous threading** and **synchronous threading**. With asynchronous threading, once the parent creates a child thread, the parent resumes its execution, so that the parent and child execute concurrently and independently of one another. Because the threads are independent, there is typically little data sharing between them. Asynchronous threading is the strategy used in the multithreaded server illustrated in Figure 4.2 and is also commonly used for designing responsive user interfaces.

Synchronous threading occurs when the parent thread creates one or more children and then must wait for all of its children to terminate before it resumes. Here, the threads created by the parent perform work concurrently, but the parent cannot continue until this work has been completed. Once each thread has finished its work, it terminates and joins with its parent. Only after all of the children have joined can the parent resume execution. Typically, synchronous threading involves significant data sharing among threads. For example, the parent thread may combine the results calculated by its various children. All of the following examples use synchronous threading.

### 4.4.1 Pthreads

**Pthreads** refers to the POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization. This is a ***specification*** for thread behavior, not an ***implementation***. Operating-system designers may implement the specification

```
#include <pthread.h>
#include <stdio.h>

#include <stdlib.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
  pthread_t tid; /* the thread identifier */
  pthread_attr_t attr; /* set of thread attributes */

  /* set the default attributes of the thread */
  pthread_attr_init(&attr);
  /* create the thread */
  pthread_create(&tid, &attr, runner, argv[1]);
  /* wait for the thread to exit */
  pthread_join(tid,NULL);

  printf("sum = %d\n",sum);
}

/* The thread will execute in this function */
void *runner(void *param)
{
  int i, upper = atoi(param);
  sum = 0;

  for (i = 1; i <= upper; i++)
    sum += i;

  pthread_exit(0);
}
```

**Figure 4.11** Multithreaded C program using the Pthreads API.

in any way they wish. Numerous systems implement the Pthreads specification; most are UNIX-type systems, including Linux and macOS. Although Windows doesn't support Pthreads natively, some third-party implementations for Windows are available.

The C program shown in Figure 4.11 demonstrates the basic Pthreads API for constructing a multithreaded program that calculates the summation of a non-negative integer in a separate thread. In a Pthreads program, separate threads begin execution in a specified function. In Figure 4.11, this is the `runner()` function. When this program begins, a single thread of control begins in

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

**Figure 4.12** Pthread code for joining ten threads.

`main()`. After some initialization, `main()` creates a second thread that begins control in the `runner()` function. Both threads share the global data `sum`.

Let's look more closely at this program. All Pthreads programs must include the `pthread.h` header file. The statement `pthread_t tid` declares the identifier for the thread we will create. Each thread has a set of attributes, including stack size and scheduling information. The `pthread_attr_t attr` declaration represents the attributes for the thread. We set the attributes in the function call `pthread_attr_init(&attr)`. Because we did not explicitly set any attributes, we use the default attributes provided. (In Chapter 5, we discuss some of the scheduling attributes provided by the Pthreads API.) A separate thread is created with the `pthread_create()` function call. In addition to passing the thread identifier and the attributes for the thread, we also pass the name of the function where the new thread will begin execution—in this case, the `runner()` function. Last, we pass the integer parameter that was provided on the command line, `argv[1]`.

At this point, the program has two threads: the initial (or parent) thread in `main()` and the summation (or child) thread performing the summation operation in the `runner()` function. This program follows the thread create/join strategy, whereby after creating the summation thread, the parent thread will wait for it to terminate by calling the `pthread_join()` function. The summation thread will terminate when it calls the function `pthread_exit()`. Once the summation thread has returned, the parent thread will output the value of the shared data `sum`.

This example program creates only a single thread. With the growing dominance of multicore systems, writing programs containing several threads has become increasingly common. A simple method for waiting on several threads using the `pthread_join()` function is to enclose the operation within a simple `for` loop. For example, you can join on ten threads using the Pthread code shown in Figure 4.12.

### 4.4.2 Windows Threads

The technique for creating threads using the Windows thread library is similar to the Pthreads technique in several ways. We illustrate the Windows thread API in the C program shown in Figure 4.13. Notice that we must include the `windows.h` header file when using the Windows API.

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* The thread will execute in this function */
DWORD WINAPI Summation(LPVOID Param)
{
  DWORD Upper = *(DWORD*)Param;
  for (DWORD i = 1; i <= Upper; i++)
    Sum += i;
  return 0;
}

int main(int argc, char *argv[])
{
  DWORD ThreadId;
  HANDLE ThreadHandle;
  int Param;

  Param = atoi(argv[1]);
  /* create the thread */
  ThreadHandle = CreateThread(
    NULL, /* default security attributes */
    0, /* default stack size */
    Summation, /* thread function */
    &Param, /* parameter to thread function */
    0, /* default creation flags */
    &ThreadId); /* returns the thread identifier */

   /* now wait for the thread to finish */
  WaitForSingleObject(ThreadHandle,INFINITE);

  /* close the thread handle */
  CloseHandle(ThreadHandle);

  printf("sum = %d\n",Sum);
}
```

**Figure 4.13** Multithreaded C program using the Windows API.

Just as in the Pthreads version shown in Figure 4.11, data shared by the separate threads—in this case, `Sum`—are declared globally (the `DWORD` data type is an unsigned 32-bit integer). We also define the `Summation()` function that is to be performed in a separate thread. This function is passed a pointer to a `void`, which Windows defines as LPVOID. The thread performing this function sets the global data `Sum` to the value of the summation from 0 to the parameter passed to `Summation()`.

Threads are created in the Windows API using the `CreateThread()` function, and—just as in Pthreads—a set of attributes for the thread is passed to this function. These attributes include security information, the size of the stack, and a flag that can be set to indicate if the thread is to start in a suspended state. In this program, we use the default values for these attributes. (The default values do not initially set the thread to a suspended state and instead make it eligible to be run by the CPU scheduler.) Once the summation thread is created, the parent must wait for it to complete before outputting the value of `Sum`, as the value is set by the summation thread. Recall that the Pthread program (Figure 4.11) had the parent thread wait for the summation thread using the `pthread_join()` statement. We perform the equivalent of this in the Windows API using the `WaitForSingleObject()` function, which causes the creating thread to block until the summation thread has exited.

In situations that require waiting for multiple threads to complete, the `WaitForMultipleObjects()` function is used. This function is passed four parameters:

1. The number of objects to wait for
2. A pointer to the array of objects
3. A flag indicating whether all objects have been signaled
4. A timeout duration (or `INFINITE`)

For example, if `THandles` is an array of thread `HANDLE` objects of size `N`, the parent thread can wait for all its child threads to complete with this statement:

```
WaitForMultipleObjects(N, THandles, TRUE, INFINITE);
```

### 4.4.3 Java Threads

Threads are the fundamental model of program execution in a Java program, and the Java language and its API provide a rich set of features for the creation and management of threads. All Java programs comprise at least a single thread of control—even a simple Java program consisting of only a `main()` method runs as a single thread in the JVM. Java threads are available on any system that provides a JVM including Windows, Linux, and macOS. The Java thread API is available for Android applications as well.

There are two techniques for explicitly creating threads in a Java program. One approach is to create a new class that is derived from the `Thread` class and to override its `run()` method. An alternative—and more commonly used—technique is to define a class that implements the `Runnable` interface. This interface defines a single abstract method with the signature `public void run()`. The code in the `run()` method of a class that implements `Runnable` is what executes in a separate thread. An example is shown below:

```
class Task implements Runnable
{
   public void run() {
      System.out.println("I am a thread.");
   }
}
```

### *LAMBDA EXPRESSIONS IN JAVA*

Beginning with Version 1.8 of the language, Java introduced Lambda expressions, which allow a much cleaner syntax for creating threads. Rather than defining a separate class that implements `Runnable`, a Lambda expression can be used instead:

```
Runnable task = () -> {
   System.out.println("I am a thread.");
};

Thread worker = new Thread(task);
worker.start();
```

Lambda expressions—as well as similar functions known as **closures**—are a prominent feature of functional programming languages and have been available in several nonfunctional languages as well including Python, C++, and C#. As we shall see in later examples in this chapter, Lamdba expressions often provide a simple syntax for developing parallel applications.

Thread creation in Java involves creating a `Thread` object and passing it an instance of a class that implements `Runnable`, followed by invoking the `start()` method on the `Thread` object. This appears in the following example:

```
Thread worker = new Thread(new Task());
worker.start();
```

Invoking the `start()` method for the new `Thread` object does two things:

1. It allocates memory and initializes a new thread in the JVM.
2. It calls the `run()` method, making the thread eligible to be run by the JVM. (Note again that we never call the `run()` method directly. Rather, we call the `start()` method, and it calls the `run()` method on our behalf.)

Recall that the parent threads in the Pthreads and Windows libraries use `pthread_join()` and `WaitForSingleObject()` (respectively) to wait for the summation threads to finish before proceeding. The `join()` method in Java provides similar functionality. (Notice that `join()` can throw an `InterruptedException`, which we choose to ignore.)

```
try {
   worker.join();
}
catch (InterruptedException ie) { }
```

If the parent must wait for several threads to finish, the `join()` method can be enclosed in a `for` loop similar to that shown for Pthreads in Figure 4.12.

#### 4.4.3.1 Java Executor Framework

Java has supported thread creation using the approach we have described thus far since its origins. However, beginning with Version 1.5 and its API, Java introduced several new concurrency features that provide developers with much greater control over thread creation and communication. These tools are available in the `java.util.concurrent` package.

Rather than explicitly creating `Thread` objects, thread creation is instead organized around the `Executor` interface:

```
public interface Executor
{
   void execute(Runnable command);
}
```

Classes implementing this interface must define the `execute()` method, which is passed a `Runnable` object. For Java developers, this means using the `Executor` rather than creating a separate `Thread` object and invoking its `start()` method. The `Executor` is used as follows:

```
Executor service = new Executor;
service.execute(new Task());
```

The `Executor` framework is based on the producer-consumer model; tasks implementing the `Runnable` interface are produced, and the threads that execute these tasks consume them. The advantage of this approach is that it not only divides thread creation from execution but also provides a mechanism for communication between concurrent tasks.

Data sharing between threads belonging to the same process occurs easily in Windows and Pthreads, since shared data are simply declared globally. As a pure object-oriented language, Java has no such notion of global data. We can pass parameters to a class that implements `Runnable`, but Java threads cannot return results. To address this need, the `java.util.concurrent` package additionally defines the `Callable` interface, which behaves similarly to `Runnable` except that a result can be returned. Results returned from `Callable` tasks are known as `Future` objects. A result can be retrieved from the `get()` method defined in the `Future` interface. The program shown in Figure 4.14 illustrates the summation program using these Java features.

The `Summation` class implements the `Callable` interface, which specifies the method `V call()`—it is the code in this `call()` method that is executed in a separate thread. To execute this code, we create a `newSingleThreadExecutor` object (provided as a static method in the `Executors` class), which is of type `ExecutorService`, and pass it a `Callable` task using its `submit()` method. (The primary difference between the `execute()` and `submit()` methods is that the former returns no result, whereas the latter returns a result as a `Future`.) Once we submit the `callable` task to the thread, we wait for its result by calling the `get()` method of the `Future` object it returns.

It is quite easy to notice at first that this model of thread creation appears more complicated than simply creating a thread and joining on its termination. However, incurring this modest degree of complication confers benefits. As we have seen, using `Callable` and `Future` allows for threads to return results.

```
import java.util.concurrent.*;

class Summation implements Callable<Integer>
{
  private int upper;
  public Summation(int upper) {
     this.upper = upper;
  }

  /* The thread will execute in this method */
  public Integer call() {
     int sum = 0;
     for (int i = 1; i <= upper; i++)
       sum += i;

     return new Integer(sum);
  }
}

public class Driver
{
 public static void main(String[] args) {
    int upper = Integer.parseInt(args[0]);

    ExecutorService pool = Executors.newSingleThreadExecutor();
    Future<Integer> result = pool.submit(new Summation(upper));

    try {
       System.out.println("sum = " + result.get());
    } catch (InterruptedException | ExecutionException ie) { }
  }
}
```

**Figure 4.14** Illustration of Java Executor framework API.

Additionally, this approach separates the creation of threads from the results they produce: rather than waiting for a thread to terminate before retrieving results, the parent instead only waits for the results to become available. Finally, as we shall see in Section 4.5.1, this framework can be combined with other features to create robust tools for managing a large number of threads.

## 4.5 Implicit Threading

With the continued growth of multicore processing, applications containing hundreds—or even thousands—of threads are looming on the horizon. Designing such applications is not a trivial undertaking: programmers must

### *THE JVM AND THE HOST OPERATING SYSTEM*

The JVM is typically implemented on top of a host operating system (see Figure 18.10). This setup allows the JVM to hide the implementation details of the underlying operating system and to provide a consistent, abstract environment that allows Java programs to operate on any platform that supports a JVM. The specification for the JVM does not indicate how Java threads are to be mapped to the underlying operating system, instead leaving that decision to the particular implementation of the JVM. For example, the Windows operating system uses the one-to-one model; therefore, each Java thread for a JVM running on Windows maps to a kernel thread. In addition, there may be a relationship between the Java thread library and the thread library on the host operating system. For example, implementations of a JVM for the Windows family of operating systems might use the Windows API when creating Java threads; Linux and macOS systems might use the Pthreads API.

address not only the challenges outlined in Section 4.2 but additional difficulties as well. These difficulties, which relate to program correctness, are covered in Chapter 6 and Chapter 8.

One way to address these difficulties and better support the design of concurrent and parallel applications is to transfer the creation and management of threading from application developers to compilers and run-time libraries. This strategy, termed **implicit threading**, is an increasingly popular trend. In this section, we explore four alternative approaches to designing applications that can take advantage of multicore processors through implicit threading. As we shall see, these strategies generally require application developers to identify ***tasks***—not threads—that can run in parallel. A task is usually written as a function, which the run-time library then maps to a separate thread, typically using the many-to-many model (Section 4.3.3). The advantage of this approach is that developers only need to identify parallel tasks, and the libraries determine the specific details of thread creation and management.

### 4.5.1 Thread Pools

In Section 4.1, we described a multithreaded web server. In this situation, whenever the server receives a request, it creates a separate thread to service the request. Whereas creating a separate thread is certainly superior to creating a separate process, a multithreaded server nonetheless has potential problems. The first issue concerns the amount of time required to create the thread, together with the fact that the thread will be discarded once it has completed its work. The second issue is more troublesome. If we allow each concurrent request to be serviced in a new thread, we have not placed a bound on the number of threads concurrently active in the system. Unlimited threads could exhaust system resources, such as CPU time or memory. One solution to this problem is to use a **thread pool**.

### *ANDROID THREAD POOLS*

In Section 3.8.2.1, we covered RPCs in the Android operating system. You may recall from that section that Android uses the Android Interface Definition Language (AIDL), a tool that specifies the remote interface that clients interact with on the server. AIDL also provides a thread pool. A remote service using the thread pool can handle multiple concurrent requests, servicing each request using a separate thread from the pool.

The general idea behind a thread pool is to create a number of threads at start-up and place them into a pool, where they sit and wait for work. When a server receives a request, rather than creating a thread, it instead submits the request to the thread pool and resumes waiting for additional requests. If there is an available thread in the pool, it is awakened, and the request is serviced immediately. If the pool contains no available thread, the task is queued until one becomes free. Once a thread completes its service, it returns to the pool and awaits more work. Thread pools work well when the tasks submitted to the pool can be executed asynchronously.

Thread pools offer these benefits:

1. Servicing a request with an existing thread is often faster than waiting to create a thread.
2. A thread pool limits the number of threads that exist at any one point. This is particularly important on systems that cannot support a large number of concurrent threads.
3. Separating the task to be performed from the mechanics of creating the task allows us to use different strategies for running the task. For example, the task could be scheduled to execute after a time delay or to execute periodically.

The number of threads in the pool can be set heuristically based on factors such as the number of CPUs in the system, the amount of physical memory, and the expected number of concurrent client requests. More sophisticated thread-pool architectures can dynamically adjust the number of threads in the pool according to usage patterns. Such architectures provide the further benefit of having a smaller pool—thereby consuming less memory—when the load on the system is low. We discuss one such architecture, Apple's Grand Central Dispatch, later in this section.

The Windows API provides several functions related to thread pools. Using the thread pool API is similar to creating a thread with the `Thread_Create()` function, as described in Section 4.4.2. Here, a function that is to run as a separate thread is defined. Such a function may appear as follows:

```
DWORD WINAPI PoolFunction(PVOID Param) {
   /* this function runs as a separate thread. */
}
```

A pointer to `PoolFunction()` is passed to one of the functions in the thread pool API, and a thread from the pool executes this function. One such member in the thread pool API is the `QueueUserWorkItem()` function, which is passed three parameters:

- `LPTHREAD_START_ROUTINE Function`—a pointer to the function that is to run as a separate thread
- `PVOID Param`—the parameter passed to `Function`
- `ULONG Flags`—flags indicating how the thread pool is to create and manage execution of the thread

An example of invoking a function is the following:

```
QueueUserWorkItem(&PoolFunction, NULL, 0);
```

This causes a thread from the thread pool to invoke `PoolFunction()` on behalf of the programmer. In this instance, we pass no parameters to `PoolFunction()`. Because we specify `0` as a flag, we provide the thread pool with no special instructions for thread creation.

Other members in the Windows thread pool API include utilities that invoke functions at periodic intervals or when an asynchronous I/O request completes.

#### 4.5.1.1 Java Thread Pools

The `java.util.concurrent` package includes an API for several varieties of thread-pool architectures. Here, we focus on the following three models:

1. Single thread executor—`newSingleThreadExecutor()`—creates a pool of size 1.
2. Fixed thread executor—`newFixedThreadPool(int size)`—creates a thread pool with a specified number of threads.
3. Cached thread executor—`newCachedThreadPool()`—creates an unbounded thread pool, reusing threads in many instances.

We have, in fact, already seen the use of a Java thread pool in Section 4.4.3, where we created a `newSingleThreadExecutor` in the program example shown in Figure 4.14. In that section, we noted that the Java executor framework can be used to construct more robust threading tools. We now describe how it can be used to create thread pools.

A thread pool is created using one of the factory methods in the `Executors` class:

- `static ExecutorService newSingleThreadExecutor()`
- `static ExecutorService newFixedThreadPool(int size)`
- `static ExecutorService newCachedThreadPool()`

Each of these factory methods creates and returns an object instance that implements the `ExecutorService` interface. `ExecutorService` extends the `Execu-`

```
import java.util.concurrent.*;

public class ThreadPoolExample
{
public static void main(String[] args) {
   int numTasks = Integer.parseInt(args[0].trim());

   /* Create the thread pool */
   ExecutorService pool = Executors.newCachedThreadPool();

   /* Run each task using a thread in the pool */
   for (int i = 0; i < numTasks; i++)
      pool.execute(new Task());

   /* Shut down the pool once all threads have completed */
   pool.shutdown();
}
```

**Figure 4.15** Creating a thread pool in Java.

tor interface, allowing us to invoke the `execute()` method on this object. In addition, `ExecutorService` provides methods for managing termination of the thread pool.

The example shown in Figure 4.15 creates a cached thread pool and submits tasks to be executed by a thread in the pool using the `execute()` method. When the `shutdown()` method is invoked, the thread pool rejects additional tasks and shuts down once all existing tasks have completed execution.

## 4.5.2 Fork Join

The strategy for thread creation covered in Section 4.4 is often known as the **fork-join** model. Recall that with this method, the main parent thread creates (*forks*) one or more child threads and then waits for the children to terminate and *join* with it, at which point it can retrieve and combine their results. This synchronous model is often characterized as explicit thread creation, but it is also an excellent candidate for implicit threading. In the latter situation, threads are not constructed directly during the fork stage; rather, parallel tasks are designated. This model is illustrated in Figure 4.16. A library manages the number of threads that are created and is also responsible for assigning tasks to threads. In some ways, this fork-join model is a synchronous version of thread pools in which a library determines the actual number of threads to create—for example, by using the heuristics described in Section 4.5.1.

### 4.5.2.1 Fork Join in Java

Java introduced a fork-join library in Version 1.7 of the API that is designed to be used with recursive divide-and-conquer algorithms such as Quicksort and Mergesort. When implementing divide-and-conquer algorithms using this

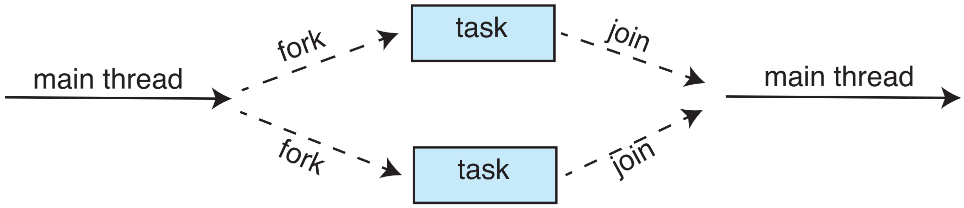


**Figure 4.16** Fork-join parallelism.

library, separate tasks are forked during the divide step and assigned smaller subsets of the original problem. Algorithms must be designed so that these separate tasks can execute concurrently. At some point, the size of the problem assigned to a task is small enough that it can be solved directly and requires creating no additional tasks. The general recursive algorithm behind Java's fork-join model is shown below:

```
Task(problem)
   if problem is small enough
      solve the problem directly
   else
      subtask1 = fork(new Task(subset of problem)
      subtask2 = fork(new Task(subset of problem)

      result1 = join(subtask1)
      result2 = join(subtask2)

      return combined results
```

Figure 4.17 depicts the model graphically.

We now illustrate Java's fork-join strategy by designing a divide-and-conquer algorithm that sums all elements in an array of integers. In Version 1.7 of the API Java introduced a new thread pool—the `ForkJoinPool`—that can be assigned tasks that inherit the abstract base class `ForkJoinTask` (which for now we will assume is the `SumTask` class). The following creates a `ForkJoinPool` object and submits the initial task via its `invoke()` method:

```
ForkJoinPool pool = new ForkJoinPool();
// array contains the integers to be summed
int[] array = new int[SIZE];

SumTask task = new SumTask(0, SIZE - 1, array);
int sum = pool.invoke(task);
```

Upon completion, the initial call to `invoke()` returns the summation of `array`.

The class `SumTask`—shown in Figure 4.18—implements a divide-and-conquer algorithm that sums the contents of the array using fork-join. New tasks are created using the `fork()` method, and the `compute()` method specifies the computation that is performed by each task. The method `compute()` is invoked until it can directly calculate the sum of the subset it is assigned. The

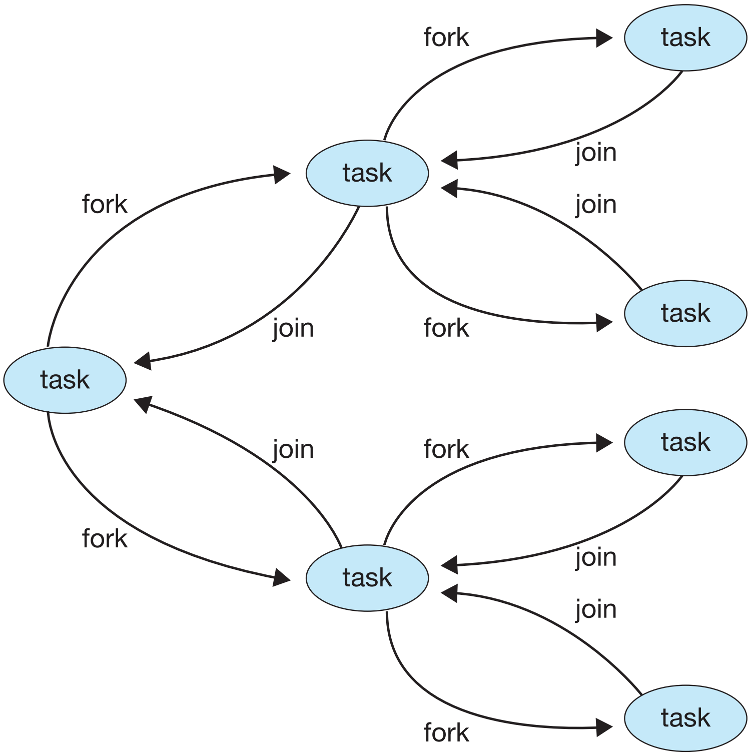


**Figure 4.17** Fork-join in Java.

call to `join()` blocks until the task completes, upon which `join()` returns the results calculated in `compute()`.

Notice that `SumTask` in Figure 4.18 extends `RecursiveTask`. The Java fork-join strategy is organized around the abstract base class `ForkJoinTask`, and the `RecursiveTask` and `RecursiveAction` classes extend this class. The fundamental difference between these two classes is that `RecursiveTask` returns a result (via the return value specified in `compute()`), and `RecursiveAction` does not return a result. The relationship between the three classes is illustrated in the UML class diagram in Figure 4.19.

An important issue to consider is determining when the problem is "small enough" to be solved directly and no longer requires creating additional tasks. In `SumTask`, this occurs when the number of elements being summed is less than the value `THRESHOLD`, which in Figure 4.18 we have arbitrarily set to 1,000. In practice, determining when a problem can be solved directly requires careful timing trials, as the value can vary according to implementation.

What is interesting in Java's fork-join model is the management of tasks wherein the library constructs a pool of worker threads and balances the load of tasks among the available workers. In some situations, there are thousands of tasks, yet only a handful of threads performing the work (for example, a separate thread for each CPU). Additionally, each thread in a `ForkJoinPool` maintains a queue of tasks that it has forked, and if a thread's queue is empty, it can steal a task from another thread's queue using a ***work stealing*** algorithm, thus balancing the workload of tasks among all threads.

```
import java.util.concurrent.*;

public class SumTask extends RecursiveTask<Integer>
{
  static final int THRESHOLD = 1000;

  private int begin;
  private int end;
  private int[] array;

  public SumTask(int begin, int end, int[] array) {
    this.begin = begin;
    this.end = end;
    this.array = array;
  }

  protected Integer compute() {
    if (end - begin < THRESHOLD) {
      int sum = 0;
      for (int i = begin; i <= end; i++)
        sum += array[i];

      return sum;
    }
    else {
      int mid = (begin + end) / 2;

      SumTask leftTask = new SumTask(begin, mid, array);
      SumTask rightTask = new SumTask(mid + 1, end, array);

      leftTask.fork();
      rightTask.fork();

      return rightTask.join() + leftTask.join();
    }
  }
}
```

**Figure 4.18** Fork-join calculation using the Java API.

### 4.5.3 OpenMP

OpenMP is a set of compiler directives as well as an API for programs written in C, C++, or FORTRAN that provides support for parallel programming in shared-memory environments. OpenMP identifies **parallel regions** as blocks of code that may run in parallel. Application developers insert compiler directives into their code at parallel regions, and these directives instruct the OpenMP run-

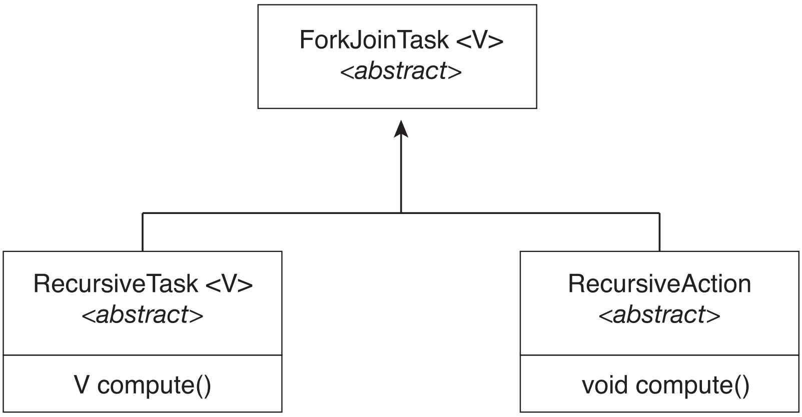


**Figure 4.19** UML class diagram for Java's fork-join.

time library to execute the region in parallel. The following C program illustrates a compiler directive above the parallel region containing the `printf()` statement:

```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  /* sequential code */

  #pragma omp parallel
  {
    printf("I am a parallel region.");
  }

  /* sequential code */

  return 0;
}
```

When OpenMP encounters the directive

```
#pragma omp parallel
```

it creates as many threads as there are processing cores in the system. Thus, for a dual-core system, two threads are created; for a quad-core system, four are created; and so forth. All the threads then simultaneously execute the parallel region. As each thread exits the parallel region, it is terminated.

OpenMP provides several additional directives for running code regions in parallel, including parallelizing loops. For example, assume we have two arrays, `a` and `b`, of size `N`. We wish to sum their contents and place the results in array `c`. We can have this task run in parallel by using the following code segment, which contains the compiler directive for parallelizing `for` loops:

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
   c[i] = a[i] + b[i];
}
```

OpenMP divides the work contained in the `for` loop among the threads it has created in response to the directive

```
#pragma omp parallel for
```

In addition to providing directives for parallelization, OpenMP allows developers to choose among several levels of parallelism. For example, they can set the number of threads manually. It also allows developers to identify whether data are shared between threads or are private to a thread. OpenMP is available on several open-source and commercial compilers for Linux, Windows, and macOS systems. We encourage readers interested in learning more about OpenMP to consult the bibliography at the end of the chapter.

### 4.5.4 Grand Central Dispatch

Grand Central Dispatch (GCD) is a technology developed by Apple for its macOS and iOS operating systems. It is a combination of a run-time library, an API, and language extensions that allow developers to identify sections of code (***tasks***) to run in parallel. Like OpenMP, GCD manages most of the details of threading.

GCD schedules tasks for run-time execution by placing them on a **dispatch queue**. When it removes a task from a queue, it assigns the task to an available thread from a pool of threads that it manages. GCD identifies two types of dispatch queues: ***serial*** and ***concurrent***.

Tasks placed on a serial queue are removed in FIFO order. Once a task has been removed from the queue, it must complete execution before another task is removed. Each process has its own serial queue (known as its **main queue**), and developers can create additional serial queues that are local to a particular process. (This is why serial queues are also known as ***private dispatch queues***.) Serial queues are useful for ensuring the sequential execution of several tasks.

Tasks placed on a concurrent queue are also removed in FIFO order, but several tasks may be removed at a time, thus allowing multiple tasks to execute in parallel. There are several system-wide concurrent queues (also known as ***global dispatch queues***), which are divided into four primary quality-of-service classes:

- `QOS_CLASS_USER_INTERACTIVE`—The **user-interactive** class represents tasks that interact with the user, such as the user interface and event handling, to ensure a responsive user interface. Completing a task belonging to this class should require only a small amount of work.
- `QOS_CLASS_USER_INITIATED`—The **user-initiated** class is similar to the user-interactive class in that tasks are associated with a responsive user interface; however, user-initiated tasks may require longer processing

times. Opening a file or a URL is a user-initiated task, for example. Tasks belonging to this class must be completed for the user to continue interacting with the system, but they do not need to be serviced as quickly as tasks in the user-interactive queue.

- `QOS_CLASS_UTILITY` —The **utility** class represents tasks that require a longer time to complete but do not demand immediate results. This class includes work such as importing data.
- `QOS_CLASS_BACKGROUND` —Tasks belonging to the **background** class are not visible to the user and are not time sensitive. Examples include indexing a mailbox system and performing backups.

Tasks submitted to dispatch queues may be expressed in one of two different ways:

1. For the C, C++, and Objective-C languages, GCD identifies a language extension known as a **block**, which is simply a self-contained unit of work. A block is specified by a caret ˆ inserted in front of a pair of braces { }. Code within the braces identifies the unit of work to be performed. A simple example of a block is shown below:

   ```
   ^{ printf("I am a block"); }
   ```

2. For the Swift programming language, a task is defined using a ***closure***, which is similar to a block in that it expresses a self-contained unit of functionality. Syntactically, a Swift closure is written in the same way as a block, minus the leading caret.

   The following Swift code segment illustrates obtaining a concurrent queue for the user-initiated class and submitting a task to the queue using the `dispatch_async()` function:

   ```
   let queue = dispatch_get_global_queue
     (QOS_CLASS_USER_INITIATED, 0)

   dispatch_async(queue,{ print("I am a closure.") })
   ```

Internally, GCD's thread pool is composed of POSIX threads. GCD actively manages the pool, allowing the number of threads to grow and shrink according to application demand and system capacity. GCD is implemented by the `libdispatch` library, which Apple has released under the Apache Commons license. It has since been ported to the FreeBSD operating system.

### 4.5.5 Intel Thread Building Blocks

Intel threading building blocks (TBB) is a template library that supports designing parallel applications in C++. As this is a library, it requires no special compiler or language support. Developers specify tasks that can run in parallel, and the TBB task scheduler maps these tasks onto underlying threads. Furthermore, the task scheduler provides load balancing and is cache aware, meaning that it will give precedence to tasks that likely have their data stored in cache memory and thus will execute more quickly. TBB provides a rich set of features, including templates for parallel loop structures, atomic operations, and mutual exclusion locking. In addition, it provides concurrent data structures, including a hash map, queue, and vector, which can serve as equivalent thread-safe versions of the C++ standard template library data structures.

Let's use parallel for loops as an example. Initially, assume there is a function named `apply(float value)` that performs an operation on the parameter `value`. If we had an array `v` of size `n` containing `float` values, we could use the following serial for loop to pass each value in `v` to the `apply()` function:

```
for (int i = 0; i < n; i++) {
  apply(v[i]);
}
```

A developer could manually apply data parallelism (Section 4.2.2) on a multicore system by assigning different regions of the array `v` to each processing core; however, this ties the technique for achieving parallelism closely to the physical hardware, and the algorithm would have to be modified and recompiled for the number of processing cores on each specific architecture.

Alternatively, a developer could use TBB, which provides a `parallel_for` template that expects two values:

`parallel_for` (***range*** ***body*** )

where ***range*** refers to the range of elements that will be iterated (known as the **iteration space**) and ***body*** specifies an operation that will be performed on a subrange of elements.

We can now rewrite the above serial for loop using the TBB `parallel_for` template as follows:

```
parallel_for (size_t(0), n, [=](size_t i) {apply(v[i]);});
```

The first two parameters specify that the iteration space is from 0 to $n-1$ (which corresponds to the number of elements in the array `v`). The second parameter is a C++ lambda function that requires a bit of explanation. The expression `[=](size_t i)` is the parameter `i`, which assumes each of the values over the iteration space (in this case from 0 to `n` − 1). Each value of `i` is used to identify which array element in `v` is to be passed as a parameter to the `apply(v[i])` function.

The TBB library will divide the loop iterations into separate "chunks" and create a number of tasks that operate on those chunks. (The `parallel_for` function allows developers to manually specify the size of the chunks if they wish to.) TBB will also create a number of threads and assign tasks to available threads. This is quite similar to the fork-join library in Java. The advantage of this approach is that it requires only that developers identify what operations can run in parallel (by specifying a `parallel_for` loop), and the library manages the details involved in dividing the work into separate tasks that run in parallel. Intel TBB has both commercial and open-source versions that run on Windows, Linux, and macOS. Refer to the bibliography for further details on how to develop parallel applications using TBB.

## 4.6 Threading Issues

In this section, we discuss some of the issues to consider in designing multithreaded programs.

### 4.6.1 The fork() and exec() System Calls

In Chapter 3, we described how the `fork()` system call is used to create a separate, duplicate process. The semantics of the `fork()` and `exec()` system calls change in a multithreaded program.

If one thread in a program calls `fork()`, does the new process duplicate all threads, or is the new process single-threaded? Some UNIX systems have chosen to have two versions of `fork()`, one that duplicates all threads and another that duplicates only the thread that invoked the `fork()` system call.

The `exec()` system call typically works in the same way as described in Chapter 3. That is, if a thread invokes the `exec()` system call, the program specified in the parameter to `exec()` will replace the entire process—including all threads.

Which of the two versions of `fork()` to use depends on the application. If `exec()` is called immediately after forking, then duplicating all threads is unnecessary, as the program specified in the parameters to `exec()` will replace the process. In this instance, duplicating only the calling thread is appropriate. If, however, the separate process does not call `exec()` after forking, the separate process should duplicate all threads.

### 4.6.2 Signal Handling

A **signal** is used in UNIX systems to notify a process that a particular event has occurred. A signal may be received either synchronously or asynchronously, depending on the source of and the reason for the event being signaled. All signals, whether synchronous or asynchronous, follow the same pattern:

1. A signal is generated by the occurrence of a particular event.
2. The signal is delivered to a process.
3. Once delivered, the signal must be handled.

Examples of synchronous signals include illegal memory access and division by 0. If a running program performs either of these actions, a signal is generated. Synchronous signals are delivered to the same process that performed the operation that caused the signal (that is the reason they are considered synchronous).

When a signal is generated by an event external to a running process, that process receives the signal asynchronously. Examples of such signals include terminating a process with specific keystrokes (such as `<control><C>`) and having a timer expire. Typically, an asynchronous signal is sent to another process.

A signal may be ***handled*** by one of two possible handlers:

1. A default signal handler
2. A user-defined signal handler

Every signal has a **default signal handler** that the kernel runs when handling that signal. This default action can be overridden by a **user-define signal handler** that is called to handle the signal. Signals are handled in different ways. Some signals may be ignored, while others (for example, an illegal memory access) are handled by terminating the program.

Handling signals in single-threaded programs is straightforward: signals are always delivered to a process. However, delivering signals is more complicated in multithreaded programs, where a process may have several threads. Where, then, should a signal be delivered?

In general, the following options exist:

1. Deliver the signal to the thread to which the signal applies.
2. Deliver the signal to every thread in the process.
3. Deliver the signal to certain threads in the process.
4. Assign a specific thread to receive all signals for the process.

The method for delivering a signal depends on the type of signal generated. For example, synchronous signals need to be delivered to the thread causing the signal and not to other threads in the process. However, the situation with asynchronous signals is not as clear. Some asynchronous signals—such as a signal that terminates a process (`<control><C>`, for example)—should be sent to all threads.

The standard UNIX function for delivering a signal is

```
kill(pid_t pid, int signal)
```

This function specifies the process (`pid`) to which a particular signal (`signal`) is to be delivered. Most multithreaded versions of UNIX allow a thread to specify which signals it will accept and which it will block. Therefore, in some cases, an asynchronous signal may be delivered only to those threads that are not blocking it. However, because signals need to be handled only once, a signal is typically delivered only to the first thread found that is not blocking it. POSIX Pthreads provides the following function, which allows a signal to be delivered to a specified thread (`tid`):

```
pthread_kill(pthread_t tid, int signal)
```

Although Windows does not explicitly provide support for signals, it allows us to emulate them using **asynchronous procedure calls** (**APCs**). The APC facility enables a user thread to specify a function that is to be called when the user thread receives notification of a particular event. As indicated by its name, an APC is roughly equivalent to an asynchronous signal in UNIX. However, whereas UNIX must contend with how to deal with signals in a multithreaded environment, the APC facility is more straightforward, since an APC is delivered to a particular thread rather than a process.

### 4.6.3 Thread Cancellation

**Thread cancellation** involves terminating a thread before it has completed. For example, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be canceled. Another situation might occur when a user presses a button on a web browser that stops a web page from loading any further. Often, a web page loads using several threads—each image is loaded in a separate thread. When a user presses the `stop` button on the browser, all threads loading the page are canceled.

A thread that is to be canceled is often referred to as the **target thread**. Cancellation of a target thread may occur in two different scenarios:

1. **Asynchronous cancellation**. One thread immediately terminates the target thread.
2. **Deferred cancellation**. The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

The difficulty with cancellation occurs in situations where resources have been allocated to a canceled thread or where a thread is canceled while in the midst of updating data it is sharing with other threads. This becomes especially troublesome with asynchronous cancellation. Often, the operating system will reclaim system resources from a canceled thread but will not reclaim all resources. Therefore, canceling a thread asynchronously may not free a necessary system-wide resource.

With deferred cancellation, in contrast, one thread indicates that a target thread is to be canceled, but cancellation occurs only after the target thread has checked a flag to determine whether or not it should be canceled. The thread can perform this check at a point at which it can be canceled safely.

In Pthreads, thread cancellation is initiated using the `pthread_cancel()` function. The identifier of the target thread is passed as a parameter to the function. The following code illustrates creating—and then canceling—a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

  . . .

/* cancel the thread */
pthread_cancel(tid);

/* wait for the thread to terminate */
pthread_join(tid,NULL);
```

Invoking `pthread_cancel()` indicates only a request to cancel the target thread, however; actual cancellation depends on how the target thread is set up to handle the request. When the target thread is finally canceled, the call to `pthread_join()` in the canceling thread returns. Pthreads supports three cancellation modes. Each mode is defined as a state and a type, as illustrated in the table below. A thread may set its cancellation state and type using an API.

| Mode | State | Type |
|---|---|---|
| Off | Disabled | – |
| Deferred | Enabled | Deferred |
| Asynchronous | Enabled | Asynchronous |

As the table illustrates, Pthreads allows threads to disable or enable cancellation. Obviously, a thread cannot be canceled if cancellation is disabled. However, cancellation requests remain pending, so the thread can later enable cancellation and respond to the request.

The default cancellation type is deferred cancellation. However, cancellation occurs only when a thread reaches a **cancellation point**. Most of the blocking system calls in the POSIX and standard C library are defined as cancellation points, and these are listed when invoking the command `man pthreads` on a Linux system. For example, the `read()` system call is a cancellation point that allows cancelling a thread that is blocked while awaiting input from `read()`.

One technique for establishing a cancellation point is to invoke the `pthread_testcancel()` function. If a cancellation request is found to be pending, the call to `pthread_testcancel()` will not return, and the thread will terminate; otherwise, the call to the function will return, and the thread will continue to run. Additionally, Pthreads allows a function known as a **cleanup handler** to be invoked if a thread is canceled. This function allows any resources a thread may have acquired to be released before the thread is terminated.

The following code illustrates how a thread may respond to a cancellation request using deferred cancellation:

```
while (1) {
   /* do some work for awhile */

     . . .

   /* check if there is a cancellation request */
   pthread_testcancel();
}
```

Because of the issues described earlier, asynchronous cancellation is not recommended in Pthreads documentation. Thus, we do not cover it here. An interesting note is that on Linux systems, thread cancellation using the Pthreads API is handled through signals (Section 4.6.2).

Thread cancellation in Java uses a policy similar to deferred cancellation in Pthreads. To cancel a Java thread, you invoke the `interrupt()` method, which sets the interruption status of the target thread to true:

```
Thread worker;

 . . .

/* set the interruption status of the thread */
worker.interrupt()
```

A thread can check its interruption status by invoking the `isInterrupted()` method, which returns a boolean value of a thread's interruption status:

```
while (!Thread.currentThread().isInterrupted()) {
    . . .
}
```

### 4.6.4 Thread-Local Storage

Threads belonging to a process share the data of the process. Indeed, this data sharing provides one of the benefits of multithreaded programming. However, in some circumstances, each thread might need its own copy of certain data. We will call such data **thread-local storage** (or **TLS**). For example, in a transaction-processing system, we might service each transaction in a separate thread. Furthermore, each transaction might be assigned a unique identifier. To associate each thread with its unique transaction identifier, we could use thread-local storage.

It is easy to confuse TLS with local variables. However, local variables are visible only during a single function invocation, whereas TLS data are visible across function invocations. Additionally, when the developer has no control over the thread creation process—for example, when using an implicit technique such as a thread pool—then an alternative approach is necessary.

In some ways, TLS is similar to `static` data; the difference is that TLS data are unique to each thread. (In fact, TLS is usually declared as `static`.) Most thread libraries and compilers provide support for TLS. For example, Java provides a `ThreadLocal<T>` class with `set()` and `get()` methods for `ThreadLocal<T>` objects. Pthreads includes the type `pthread_key_t`, which provides a key that is specific to each thread. This key can then be used to access TLS data. Microsoft's C# language simply requires adding the storage attribute `[ThreadStatic]` to declare thread-local data. The `gcc` compiler provides the storage class keyword `__thread` for declaring TLS data. For example, if we wished to assign a unique identifier for each thread, we would declare it as follows:

```
static __thread int threadID;
```

### 4.6.5 Scheduler Activations

A final issue to be considered with multithreaded programs concerns communication between the kernel and the thread library, which may be required by the many-to-many and two-level models discussed in Section 4.3.3. Such coordination allows the number of kernel threads to be dynamically adjusted to help ensure the best performance.

Many systems implementing either the many-to-many or the two-level model place an intermediate data structure between the user and kernel threads. This data structure—typically known as a **lightweight process**, or **LWP**—is shown in Figure 4.20. To the user-thread library, the LWP appears to be a virtual processor on which the application can schedule a user thread to run. Each LWP is attached to a kernel thread, and it is kernel threads that the operating system schedules to run on physical processors. If a kernel thread blocks (such as while waiting for an I/O operation to complete), the LWP blocks as well. Up the chain, the user-level thread attached to the LWP also blocks.

An application may require any number of LWPs to run efficiently. Consider a CPU-bound application running on a single processor. In this scenario, only one thread can run at a time, so one LWP is sufficient. An application that is I/O-intensive may require multiple LWPs to execute, however. Typically, an LWP is required for each concurrent blocking system call. Suppose, for example, that five different file-read requests occur simultaneously. Five LWPs are needed, because all could be waiting for I/O completion in the kernel. If a process has only four LWPs, then the fifth request must wait for one of the LWPs to return from the kernel.

One scheme for communication between the user-thread library and the kernel is known as **scheduler activation**. It works as follows: The kernel provides an application with a set of virtual processors (LWPs), and the application can schedule user threads onto an available virtual processor. Furthermore, the kernel must inform an application about certain events. This procedure is known as an **upcall**. Upcalls are handled by the thread library with an **upcall handler**, and upcall handlers must run on a virtual processor.

One event that triggers an upcall occurs when an application thread is about to block. In this scenario, the kernel makes an upcall to the application informing it that a thread is about to block and identifying the specific thread. The kernel then allocates a new virtual processor to the application. The application runs an upcall handler on this new virtual processor, which saves the

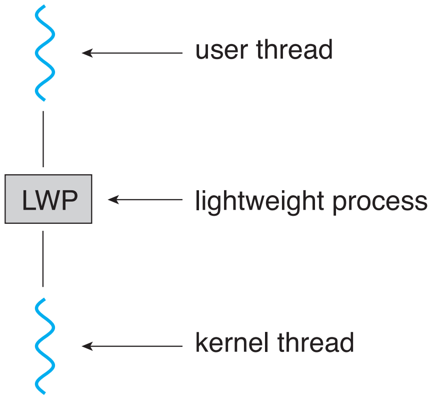


**Figure 4.20** Lightweight process (LWP).

state of the blocking thread and relinquishes the virtual processor on which the blocking thread is running. The upcall handler then schedules another thread that is eligible to run on the new virtual processor. When the event that the blocking thread was waiting for occurs, the kernel makes another upcall to the thread library informing it that the previously blocked thread is now eligible to run. The upcall handler for this event also requires a virtual processor, and the kernel may allocate a new virtual processor or preempt one of the user threads and run the upcall handler on its virtual processor. After marking the unblocked thread as eligible to run, the application schedules an eligible thread to run on an available virtual processor.

## 4.7 Operating-System Examples

At this point, we have examined a number of concepts and issues related to threads. We conclude the chapter by exploring how threads are implemented in Windows and Linux systems.

### 4.7.1 Windows Threads

A Windows application runs as a separate process, and each process may contain one or more threads. The Windows API for creating threads is covered in Section 4.4.2. Additionally, Windows uses the one-to-one mapping described in Section 4.3.2, where each user-level thread maps to an associated kernel thread.

The general components of a thread include:

- A thread ID uniquely identifying the thread
- A register set representing the status of the processor
- A program counter
- A user stack, employed when the thread is running in user mode, and a kernel stack, employed when the thread is running in kernel mode
- A private storage area used by various run-time libraries and dynamic link libraries (DLLs)

The register set, stacks, and private storage area are known as the **context** of the thread.

The primary data structures of a thread include:

- ETHREAD—executive thread block
- KTHREAD—kernel thread block
- TEB—thread environment block

The key components of the ETHREAD include a pointer to the process to which the thread belongs and the address of the routine in which the thread starts control. The ETHREAD also contains a pointer to the corresponding KTHREAD.

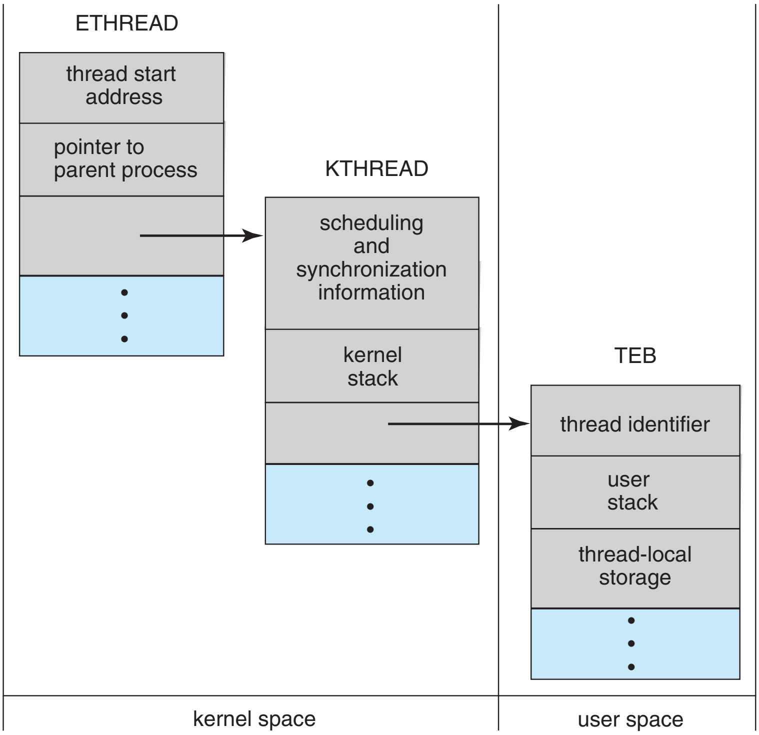

ETHREAD
thread start address
pointer to parent process
KTHREAD
scheduling and synchronization information
kernel stack
TEB
thread identifier
user stack
thread-local storage
kernel space
user space

**Figure 4.21** Data structures of a Windows thread.

The KTHREAD includes scheduling and synchronization information for the thread. In addition, the KTHREAD includes the kernel stack (used when the thread is running in kernel mode) and a pointer to the TEB.

The ETHREAD and the KTHREAD exist entirely in kernel space; this means that only the kernel can access them. The TEB is a user-space data structure that is accessed when the thread is running in user mode. Among other fields, the TEB contains the thread identifier, a user-mode stack, and an array for thread-local storage. The structure of a Windows thread is illustrated in Figure 4.21.

### 4.7.2 Linux Threads

Linux provides the `fork()` system call with the traditional functionality of duplicating a process, as described in Chapter 3. Linux also provides the ability to create threads using the `clone()` system call. However, Linux does not distinguish between processes and threads. In fact, Linux uses the term ***task*** —rather than ***process*** or ***thread***— when referring to a flow of control within a program.

When `clone()` is invoked, it is passed a set of flags that determine how much sharing is to take place between the parent and child tasks. Some of these flags are listed in Figure 4.22. For example, suppose that `clone()` is passed the flags `CLONE_FS`, `CLONE_VM`, `CLONE_SIGHAND`, and `CLONE_FILES`. The parent and child tasks will then share the same file-system information (such as the current working directory), the same memory space, the same signal handlers,

| flag | meaning |
|---|---|
| `CLONE_FS` | File-system information is shared. |
| `CLONE_VM` | The same memory space is shared. |
| `CLONE_SIGHAND` | Signal handlers are shared. |
| `CLONE_FILES` | The set of open files is shared. |

**Figure 4.22** Some of the flags passed when `clone()` is invoked.

and the same set of open files. Using `clone()` in this fashion is equivalent to creating a thread as described in this chapter, since the parent task shares most of its resources with its child task. However, if none of these flags is set when `clone()` is invoked, no sharing takes place, resulting in functionality similar to that provided by the `fork()` system call.

The varying level of sharing is possible because of the way a task is represented in the Linux kernel. A unique kernel data structure (specifically, `struct task_struct`) exists for each task in the system. This data structure, instead of storing data for the task, contains pointers to other data structures where these data are stored—for example, data structures that represent the list of open files, signal-handling information, and virtual memory. When `fork()` is invoked, a new task is created, along with a ***copy*** of all the associated data structures of the parent process. A new task is also created when the `clone()` system call is made. However, rather than copying all data structures, the new task ***points*** to the data structures of the parent task, depending on the set of flags passed to `clone()`.

Finally, the flexibility of the `clone()` system call can be extended to the concept of containers, a virtualization topic which was introduced in Chapter 1. Recall from that chapter that a container is a virtualization technique provided by the operating system that allows creating multiple Linux systems (containers) under a single Linux kernel that run in isolation to one another. Just as certain flags passed to `clone()` can distinguish between creating a task that behaves more like a process or a thread based upon the amount of sharing between the parent and child tasks, there are other flags that can be passed to `clone()` that allow a Linux container to be created. Containers will be covered more fully in Chapter 18.

## 4.8 Summary

- A thread represents a basic unit of CPU utilization, and threads belonging to the same process share many of the process resources, including code and data.
- There are four primary benefits to multithreaded applications: (1) responsiveness, (2) resource sharing, (3) economy, and (4) scalability.
- Concurrency exists when multiple threads are making progress, whereas parallelism exists when multiple threads are making progress simulta-

neously. On a system with a single CPU, only concurrency is possible; parallelism requires a multicore system that provides multiple CPUs.

- There are several challenges in designing multithreaded applications. They include dividing and balancing the work, dividing the data between the different threads, and identifying any data dependencies. Finally, multithreaded programs are especially challenging to test and debug.
- Data parallelism distributes subsets of the same data across different computing cores and performs the same operation on each core. Task parallelism distributes not data but tasks across multiple cores. Each task is running a unique operation.
- User applications create user-level threads, which must ultimately be mapped to kernel threads to execute on a CPU. The many-to-one model maps many user-level threads to one kernel thread. Other approaches include the one-to-one and many-to-many models.
- A thread library provides an API for creating and managing threads. Three common thread libraries include Windows, Pthreads, and Java threading. Windows is for the Windows system only, while Pthreads is available for POSIX-compatible systems such as UNIX, Linux, and macOS. Java threads will run on any system that supports a Java virtual machine.
- Implicit threading involves identifying tasks—not threads—and allowing languages or API frameworks to create and manage threads. There are several approaches to implicit threading, including thread pools, fork-join frameworks, and Grand Central Dispatch. Implicit threading is becoming an increasingly common technique for programmers to use in developing concurrent and parallel applications.
- Threads may be terminated using either asynchronous or deferred cancellation. Asynchronous cancellation stops a thread immediately, even if it is in the middle of performing an update. Deferred cancellation informs a thread that it should terminate but allows the thread to terminate in an orderly fashion. In most circumstances, deferred cancellation is preferred to asynchronous termination.
- Unlike many other operating systems, Linux does not distinguish between processes and threads; instead, it refers to each as a task. The Linux `clone()` system call can be used to create tasks that behave either more like processes or more like threads.

## Practice Exercises

**4.1** Provide three programming examples in which multithreading provides better performance than a single-threaded solution.

**4.2** Using Amdahl’s Law, calculate the speedup gain of an application that has a 60 percent parallel component for (a) two processing cores and (b) four processing cores.

**4.3** Does the multithreaded web server described in Section 4.1 exhibit task or data parallelism?

**4.4** What are two differences between user-level threads and kernel-level threads? Under what circumstances is one type better than the other?

**4.5** Describe the actions taken by a kernel to context-switch between kernel-level threads.

**4.6** What resources are used when a thread is created? How do they differ from those used when a process is created?

**4.7** Assume that an operating system maps user-level threads to the kernel using the many-to-many model and that the mapping is done through LWPs. Furthermore, the system allows developers to create real-time threads for use in real-time systems. Is it necessary to bind a real-time thread to an LWP? Explain.

## Further Reading

[Vahalia (1996)] covers threading in several versions of UNIX. [McDougall and Mauro (2007)] describes developments in threading the Solaris kernel. [Russinovich et al. (2017)] discuss threading in the Windows operating system family. [Mauerer (2008)] and [Love (2010)] explain how Linux handles threading, and [Levin (2013)] covers threads in macOS and iOS. [Herlihy and Shavit (2012)] covers parallelism issues on multicore systems. [Aubanel (2017)] covers parallelism of several different algorithms.

## Bibliography

**[Aubanel (2017)]** E. Aubanel, *Elements of Parallel Computing*, CRC Press (2017).

**[Herlihy and Shavit (2012)]** M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming,* Revised First Edition, Morgan Kaufmann Publishers Inc. (2012).

**[Levin (2013)]** J. Levin, *Mac OS X and iOS Internals to the Apple's Core*, Wiley (2013).

**[Love (2010)]** R. Love, *Linux Kernel Development*, Third Edition, Developer's Library (2010).

**[Mauerer (2008)]** W. Mauerer, *Professional Linux Kernel Architecture*, John Wiley and Sons (2008).

**[McDougall and Mauro (2007)]** R. McDougall and J. Mauro, *Solaris Internals,* Second Edition, Prentice Hall (2007).

**[Russinovich et al. (2017)]** M. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals–Part 1,* Seventh Edition, Microsoft Press (2017).

**[Vahalia (1996)]** U. Vahalia, *Unix Internals: The New Frontiers*, Prentice Hall (1996).

## Chapter 4 Exercises

**4.8** Provide two programming examples in which multithreading does ***not*** provide better performance than a single-threaded solution.

**4.9** Under what circumstances does a multithreaded solution using multiple kernel threads provide better performance than a single-threaded solution on a single-processor system?

**4.10** Which of the following components of program state are shared across threads in a multithreaded process?

a. Register values

b. Heap memory

c. Global variables

d. Stack memory

**4.11** Can a multithreaded solution using multiple user-level threads achieve better performance on a multiprocessor system than on a single-processor system? Explain.

**4.12** In Chapter 3, we discussed Google's Chrome browser and its practice of opening each new tab in a separate process. Would the same benefits have been achieved if, instead, Chrome had been designed to open each new tab in a separate thread? Explain.

**4.13** Is it possible to have concurrency but not parallelism? Explain.

**4.14** Using Amdahl's Law, calculate the speedup gain for the following applications:

- 40 percent parallel with (a) eight processing cores and (b) sixteen processing cores
- 67 percent parallel with (a) two processing cores and (b) four processing cores
- 90 percent parallel with (a) four processing cores and (b) eight processing cores

**4.15** Determine if the following problems exhibit task or data parallelism:

- Using a separate thread to generate a thumbnail for each photo in a collection
- Transposing a matrix in parallel
- A networked application where one thread reads from the network and another writes to the network
- The fork-join array summation application described in Section 4.5.2
- The Grand Central Dispatch system

**4.16** A system with two dual-core processors has four processors available for scheduling. A CPU-intensive application is running on this system. All input is performed at program start-up, when a single file must be opened. Similarly, all output is performed just before the program terminates, when the program results must be written to a single file. Between start-up and termination, the program is entirely CPU-bound. Your task is to improve the performance of this application by multithreading it. The application runs on a system that uses the one-to-one threading model (each user thread maps to a kernel thread).

- How many threads will you create to perform the input and output? Explain.
- How many threads will you create for the CPU-intensive portion of the application? Explain.

**4.17** Consider the following code segment:

```
pid_t pid;

pid = fork();
if (pid == 0) { /* child process */
   fork();
   thread_create( . . .);
}
fork();
```

a. How many unique processes are created?

b. How many unique threads are created?

**4.18** As described in Section 4.7.2, Linux does not distinguish between processes and threads. Instead, Linux treats both in the same way, allowing a task to be more akin to a process or a thread depending on the set of flags passed to the `clone()` system call. However, other operating systems, such as Windows, treat processes and threads differently. Typically, such systems use a notation in which the data structure for a process contains pointers to the separate threads belonging to the process. Contrast these two approaches for modeling processes and threads within the kernel.

**4.19** The program shown in Figure 4.23 uses the Pthreads API. What would be the output from the program at `LINE C` and `LINE P`?

**4.20** Consider a multicore system and a multithreaded program written using the many-to-many threading model. Let the number of user-level threads in the program be greater than the number of processing cores in the system. Discuss the performance implications of the following scenarios.

a. The number of kernel threads allocated to the program is less than the number of processing cores.

b. The number of kernel threads allocated to the program is equal to the number of processing cores.

c. The number of kernel threads allocated to the program is greater than the number of processing cores but less than the number of user-level threads.

```
#include <pthread.h>
#include <stdio.h>

int value = 0;
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
pid_t pid;
pthread_t tid;
pthread_attr_t attr;

  pid = fork();

  if (pid == 0) { /* child process */
    pthread_attr_init(&attr);
    pthread_create(&tid,&attr,runner,NULL);
    pthread_join(tid,NULL);
    printf("CHILD: value = %d",value); /* LINE C */
  }
  else if (pid > 0) { /* parent process */
    wait(NULL);
    printf("PARENT: value = %d",value); /* LINE P */
  }
}

void *runner(void *param) {
  value = 5;
  pthread_exit(0);
}
```

**Figure 4.22** C program for Exercise 4.19.

**4.21** Pthreads provides an API for managing thread cancellation. The `pthread_setcancelstate()` function is used to set the cancellation state. Its prototype appears as follows:

```
pthread_setcancelstate(int state, int *oldstate)
```

The two possible values for the state are `PTHREAD_CANCEL_ENABLE` and `PTHREAD_CANCEL_DISABLE`.

Using the code segment shown in Figure 4.24, provide examples of two operations that would be suitable to perform between the calls to disable and enable thread cancellation.

```
int oldstate;

pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &oldstate);

/* What operations would be performed here? */

pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &oldstate);
```

**Figure 4.23** C program for Exercise 4.21.

## Programming Problems

**4.22** Write a multithreaded program that calculates various statistical values for a list of numbers. This program will be passed a series of numbers on the command line and will then create three separate worker threads. One thread will determine the average of the numbers, the second will determine the maximum value, and the third will determine the minimum value. For example, suppose your program is passed the integers

90 81 78 95 79 72 85

The program will report

```
The average value is 82
The minimum value is 72
The maximum value is 95
```

The variables representing the average, minimum, and maximum values will be stored globally. The worker threads will set these values, and the parent thread will output the values once the workers have exited. (We could obviously expand this program by creating additional threads that determine other statistical values, such as median and standard deviation.)

**4.23** Write a multithreaded program that outputs prime numbers. This program should work as follows: The user will run the program and will enter a number on the command line. The program will then create a separate thread that outputs all the prime numbers less than or equal to the number entered by the user.

**4.24** An interesting way of calculating $\pi$ is to use a technique known as ***Monte Carlo***, which involves randomization. This technique works as follows:

Suppose you have a circle inscribed within a square, as shown in Figure 4.25. (Assume that the radius of this circle is 1.)

- First, generate a series of random points as simple $(x, y)$ coordinates. These points must fall within the Cartesian coordinates that bound the square. Of the total number of random points that are generated, some will occur within the circle.
- Next, estimate $\pi$ by performing the following calculation:

$$\pi = 4 \times \textit{(number of points in circle)} \ / \ \textit{(total number of points)}$$

Write a multithreaded version of this algorithm that creates a separate thread to generate a number of random points. The thread will count the number of points that occur within the circle and store that result in a global variable. When this thread has exited, the parent thread will calculate and output the estimated value of $\pi$. It is worth experimenting with the number of random points generated. As a general rule, the greater the number of points, the closer the approximation to $\pi$.

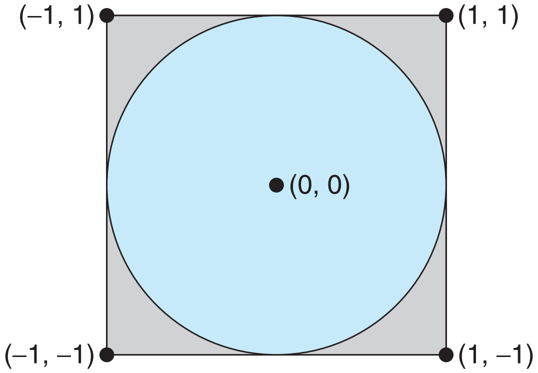


**Figure 4.25** Monte Carlo technique for calculating π.

In the source-code download for this text, you will find a sample program that provides a technique for generating random numbers, as well as determining if the random $(x, y)$ point occurs within the circle.

Readers interested in the details of the Monte Carlo method for estimating π should consult the bibliography at the end of this chapter. In Chapter 6, we modify this exercise using relevant material from that chapter.

**4.25** Repeat Exercise 4.24, but instead of using a separate thread to generate random points, use OpenMP to parallelize the generation of points. Be careful not to place the calculation of π in the parallel region, since you want to calculate π only once.

**4.26** Modify the socket-based date server (Figure 3.27) in Chapter 3 so that the server services each client request in a separate thread.

**4.27** The Fibonacci sequence is the series of numbers $0, 1, 1, 2, 3, 5, 8, \ldots$. Formally, it can be expressed as:

$$\begin{aligned} fib_0 &= 0 \\ fib_1 &= 1 \\ fib_n &= fib_{n-1} + fib_{n-2} \end{aligned}$$

Write a multithreaded program that generates the Fibonacci sequence. This program should work as follows: On the command line, the user will enter the number of Fibonacci numbers that the program is to generate. The program will then create a separate thread that will generate the Fibonacci numbers, placing the sequence in data that can be shared by the threads (an array is probably the most convenient data structure). When the thread finishes execution, the parent thread will output the sequence generated by the child thread. Because the parent thread cannot begin outputting the Fibonacci sequence until the child thread finishes, the parent thread will have to wait for the child thread to finish. Use the techniques described in Section 4.4 to meet this requirement.

**4.28** Modify programming problem Exercise 3.20 from Chapter 3, which asks you to design a pid manager. This modification will consist of writing a multithreaded program that tests your solution to Exercise 3.20. You will create a number of threads—for example, 100—and each thread will request a pid, sleep for a random period of time, and then release the pid. (Sleeping for a random period of time approximates the typical pid usage in which a pid is assigned to a new process, the process executes and then terminates, and the pid is released on the process's termination.) On UNIX and Linux systems, sleeping is accomplished through the `sleep()` function, which is passed an integer value representing the number of seconds to sleep. This problem will be modified in Chapter 7.

**4.29** Exercise 3.25 in Chapter 3 involves designing an echo server using the Java threading API. This server is single-threaded, meaning that the server cannot respond to concurrent echo clients until the current client exits. Modify the solution to Exercise 3.25 so that the echo server services each client in a separate request.

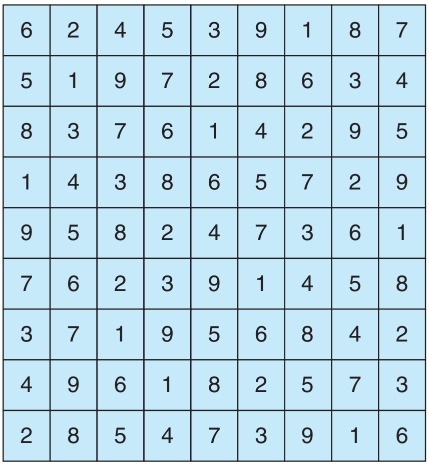
## Programming Projects

### Project 1—Sudoku Solution Validator

A ***Sudoku*** puzzle uses a 9 × 9 grid in which each column and row, as well as each of the nine 3 × 3 subgrids, must contain all of the digits 1 · · · 9. Figure 4.26 presents an example of a valid Sudoku puzzle. This project consists of designing a multithreaded application that determines whether the solution to a Sudoku puzzle is valid.

There are several different ways of multithreading this application. One suggested strategy is to create threads that check the following criteria:

- A thread to check that each column contains the digits 1 through 9
- A thread to check that each row contains the digits 1 through 9



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 6 | 2 | 4 | 5 | 3 | 9 | 1 | 8 | 7 |
| 5 | 1 | 9 | 7 | 2 | 8 | 6 | 3 | 4 |
| 8 | 3 | 7 | 6 | 1 | 4 | 2 | 9 | 5 |
| 1 | 4 | 3 | 8 | 6 | 5 | 7 | 2 | 9 |
| 9 | 5 | 8 | 2 | 4 | 7 | 3 | 6 | 1 |
| 7 | 6 | 2 | 3 | 9 | 1 | 4 | 5 | 8 |
| 3 | 7 | 1 | 9 | 5 | 6 | 8 | 4 | 2 |
| 4 | 9 | 6 | 1 | 8 | 2 | 5 | 7 | 3 |
| 2 | 8 | 5 | 4 | 7 | 3 | 9 | 1 | 6 |

**Figure 4.26** Solution to a 9 × 9 Sudoku puzzle.

- Nine threads to check that each of the 3 × 3 subgrids contains the digits 1 through 9

This would result in a total of eleven separate threads for validating a Sudoku puzzle. However, you are welcome to create even more threads for this project. For example, rather than creating one thread that checks all nine columns, you could create nine separate threads and have each of them check one column.

### I. Passing Parameters to Each Thread

The parent thread will create the worker threads, passing each worker the location that it must check in the Sudoku grid. This step will require passing several parameters to each thread. The easiest approach is to create a data structure using a `struct`. For example, a structure to pass the row and column where a thread must begin validating would appear as follows:

```
/* structure for passing data to threads */
typedef struct
{
   int row;
   int column;
} parameters;
```

Both Pthreads and Windows programs will create worker threads using a strategy similar to that shown below:

```
parameters *data = (parameters *) malloc(sizeof(parameters));
data->row = 1;
data->column = 1;
/* Now create the thread passing it data as a parameter */
```

The `data` pointer will be passed to either the `pthread_create()` (Pthreads) function or the `CreateThread()` (Windows) function, which in turn will pass it as a parameter to the function that is to run as a separate thread.

### II. Returning Results to the Parent Thread

Each worker thread is assigned the task of determining the validity of a particular region of the Sudoku puzzle. Once a worker has performed this check, it must pass its results back to the parent. One good way to handle this is to create an array of integer values that is visible to each thread. The $i^{th}$ index in this array corresponds to the $i^{th}$ worker thread. If a worker sets its corresponding value to 1, it is indicating that its region of the Sudoku puzzle is valid. A value of 0 indicates otherwise. When all worker threads have completed, the parent thread checks each entry in the result array to determine if the Sudoku puzzle is valid.

## Project 2—Multithreaded Sorting Application

Write a multithreaded sorting program that works as follows: A list of integers is divided into two smaller lists of equal size. Two separate threads (which we will term ***sorting threads***) sort each sublist using a sorting algorithm of your choice. The two sublists are then merged by a third thread—a ***merging thread*** —which merges the two sublists into a single sorted list.

Because global data are shared across all threads, perhaps the easiest way to set up the data is to create a global array. Each sorting thread will work on one half of this array. A second global array of the same size as the unsorted integer array will also be established. The merging thread will then merge the two sublists into this second array. Graphically, this program is structured as in Figure 4.27.

This programming project will require passing parameters to each of the sorting threads. In particular, it will be necessary to identify the starting index from which each thread is to begin sorting. Refer to the instructions in Project 1 for details on passing parameters to a thread.

The parent thread will output the sorted array once all sorting threads have exited.

## Project 3—Fork-Join Sorting Application

Implement the preceding project (Multithreaded Sorting Application) using Java's fork-join parallelism API. This project will be developed in two different versions. Each version will implement a different divide-and-conquer sorting algorithm:

1. Quicksort
2. Mergesort

The Quicksort implementation will use the Quicksort algorithm for dividing the list of elements to be sorted into a *left half* and a *right half* based on the

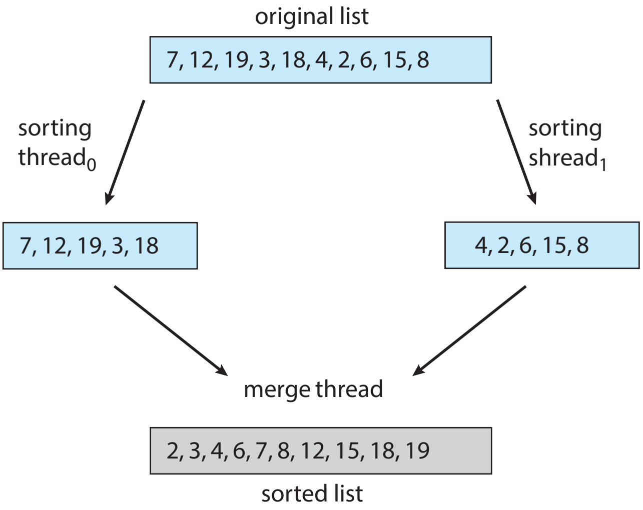


**Figure 4.27** Multithreaded sorting.

position of the pivot value. The Mergesort algorithm will divide the list into two evenly sized halves. For both the Quicksort and Mergesort algorithms, when the list to be sorted falls within some threshold value (for example, the list is size 100 or fewer), directly apply a simple algorithm such as the Selection or Insertion sort. Most data structures texts describe these two well-known, divide-and-conquer sorting algorithms.

The class `SumTask` shown in Section 4.5.2.1 extends `RecursiveTask`, which is a result-bearing `ForkJoinTask`. As this assignment will involve sorting the array that is passed to the task, but not returning any values, you will instead create a class that extends `RecursiveAction`, a non result-bearing `ForkJoinTask` (see Figure 4.19).

The objects passed to each sorting algorithm are required to implement Java's `Comparable` interface, and this will need to be reflected in the class definition for each sorting algorithm. The source code download for this text includes Java code that provides the foundations for beginning this project.

CHAPTER 5

# CPU Scheduling

CPU scheduling is the basis of multiprogrammed operating systems. By switching the CPU among processes, the operating system can make the computer more productive. In this chapter, we introduce basic CPU-scheduling concepts and present several CPU-scheduling algorithms, including real-time systems. We also consider the problem of selecting an algorithm for a particular system.

In Chapter 4, we introduced threads to the process model. On modern operating systems it is kernel-level threads—not processes—that are in fact being scheduled by the operating system. However, the terms "process scheduling" and "thread scheduling" are often used interchangeably. In this chapter, we use ***process scheduling*** when discussing general scheduling concepts and ***thread scheduling*** to refer to thread-specific ideas.

Similarly, in Chapter 1 we describe how a ***core*** is the basic computational unit of a CPU, and that a process executes on a CPU's core. However, in many instances in this chapter, when we use the general terminology of scheduling a process to "run on a CPU", we are implying that the process is running on a CPU's core.

## CHAPTER OBJECTIVES

- Describe various CPU scheduling algorithms.
- Assess CPU scheduling algorithms based on scheduling criteria.
- Explain the issues related to multiprocessor and multicore scheduling.
- Describe various real-time scheduling algorithms.
- Describe the scheduling algorithms used in the Windows, Linux, and Solaris operating systems.
- Apply modeling and simulations to evaluate CPU scheduling algorithms.
- Design a program that implements several different CPU scheduling algorithms.

## 5.1 Basic Concepts

In a system with a single CPU core, only one process can run at a time. Others must wait until the CPU's core is free and can be rescheduled. The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The idea is relatively simple. A process is executed until it must wait, typically for the completion of some I/O request. In a simple computer system, the CPU then just sits idle. All this waiting time is wasted; no useful work is accomplished. With multiprogramming, we try to use this time productively. Several processes are kept in memory at one time. When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues. Every time one process has to wait, another process can take over use of the CPU. On a multicore system, this concept of keeping the CPU busy is extended to all processing cores on the system.

Scheduling of this kind is a fundamental operating-system function. Almost all computer resources are scheduled before use. The CPU is, of course, one of the primary computer resources. Thus, its scheduling is central to operating-system design.

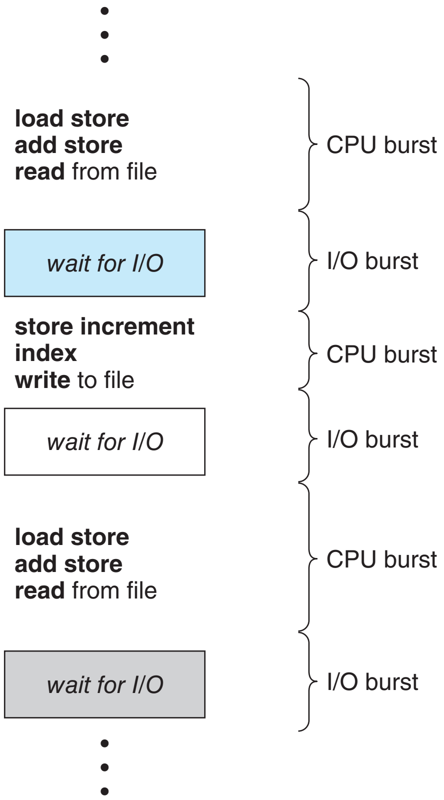

load store
add store
read from file
CPU burst
wait for I/O
I/O burst
store increment
index
write to file
CPU burst
wait for I/O
I/O burst
load store
add store
read from file
CPU burst
wait for I/O
I/O burst

**Figure 5.1** Alternating sequence of CPU and I/O bursts.

### 5.1.1 CPU–I/O Burst Cycle

The success of CPU scheduling depends on an observed property of processes: process execution consists of a **cycle** of CPU execution and I/O wait. Processes alternate between these two states. Process execution begins with a **CPU burst**. That is followed by an **I/O burst**, which is followed by another CPU burst, then another I/O burst, and so on. Eventually, the final CPU burst ends with a system request to terminate execution (Figure 5.1).

The durations of CPU bursts have been measured extensively. Although they vary greatly from process to process and from computer to computer, they tend to have a frequency curve similar to that shown in Figure 5.2. The curve is generally characterized as exponential or hyperexponential, with a large number of short CPU bursts and a small number of long CPU bursts. An I/O-bound program typically has many short CPU bursts. A CPU-bound program might have a few long CPU bursts. This distribution can be important when implementing a CPU-scheduling algorithm.

### 5.1.2 CPU Scheduler

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the **CPU scheduler**, which selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.

Note that the ready queue is not necessarily a first-in, first-out (FIFO) queue. As we shall see when we consider the various scheduling algorithms, a ready queue can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list. Conceptually, however, all the processes in the ready queue are lined up waiting for a chance to run on the CPU. The records in the queues are generally process control blocks (PCBs) of the processes.

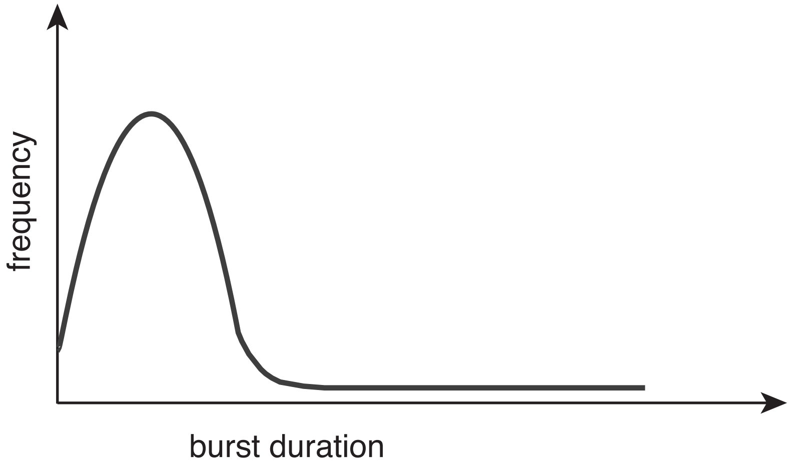


**Figure 5.2** Histogram of CPU-burst durations.

### 5.1.3 Preemptive and Nonpreemptive Scheduling

CPU-scheduling decisions may take place under the following four circumstances:

1. When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of `wait()` for the termination of a child process)
2. When a process switches from the running state to the ready state (for example, when an interrupt occurs)
3. When a process switches from the waiting state to the ready state (for example, at completion of I/O)
4. When a process terminates

For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. There is a choice, however, for situations 2 and 3.

When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is **nonpreemptive** or **cooperative**. Otherwise, it is **preemptive**. Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases it either by terminating or by switching to the waiting state. Virtually all modern operating systems including Windows, macOS, Linux, and UNIX use preemptive scheduling algorithms.

Unfortunately, preemptive scheduling can result in race conditions when data are shared among several processes. Consider the case of two processes that share data. While one process is updating the data, it is preempted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state. This issue will be explored in detail in Chapter 6.

Preemption also affects the design of the operating-system kernel. During the processing of a system call, the kernel may be busy with an activity on behalf of a process. Such activities may involve changing important kernel data (for instance, I/O queues). What happens if the process is preempted in the middle of these changes and the kernel (or the device driver) needs to read or modify the same structure? Chaos ensues. As will be discussed in Section 6.2, operating-system kernels can be designed as either nonpreemptive or preemptive. A nonpreemptive kernel will wait for a system call to complete or for a process to block while waiting for I/O to complete to take place before doing a context switch. This scheme ensures that the kernel structure is simple, since the kernel will not preempt a process while the kernel data structures are in an inconsistent state. Unfortunately, this kernel-execution model is a poor one for supporting real-time computing, where tasks must complete execution within a given time frame. In Section 5.6, we explore scheduling demands of real-time systems. A preemptive kernel requires mechanisms such as mutex locks to prevent race conditions when accessing shared kernel data structures. Most modern operating systems are now fully preemptive when running in kernel mode.

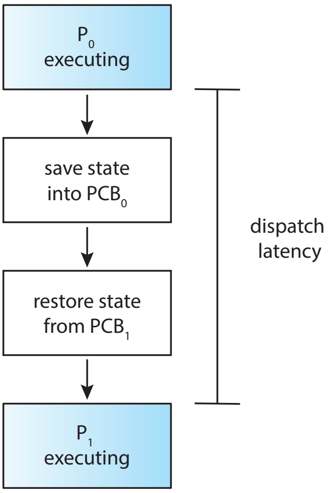


**Figure 5.3** The role of the dispatcher.

Because interrupts can, by definition, occur at any time, and because they cannot always be ignored by the kernel, the sections of code affected by interrupts must be guarded from simultaneous use. The operating system needs to accept interrupts at almost all times. Otherwise, input might be lost or output overwritten. So that these sections of code are not accessed concurrently by several processes, they disable interrupts at entry and reenable interrupts at exit. It is important to note that sections of code that disable interrupts do not occur very often and typically contain few instructions.

### 5.1.4 Dispatcher

Another component involved in the CPU-scheduling function is the **dispatcher**. The dispatcher is the module that gives control of the CPU's core to the process selected by the CPU scheduler. This function involves the following:

- Switching context from one process to another
- Switching to user mode
- Jumping to the proper location in the user program to resume that program

The dispatcher should be as fast as possible, since it is invoked during every context switch. The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency** and is illustrated in Figure 5.3.

An interesting question to consider is, how often do context switches occur? On a system-wide level, the number of context switches can be obtained by using the `vmstat` command that is available on Linux systems. Below is the output (which has been trimmed) from the command

```
vmstat 1 3
```

This command provides 3 lines of output over a 1-second delay:

```
------cpu-----
24
225
339
```

The first line gives the average number of context switches over 1 second since the system booted, and the next two lines give the number of context switches over two 1-second intervals. Since this machine booted, it has averaged 24 context switches per second. And in the past second, 225 context switches were made, with 339 context switches in the second prior to that.

We can also use the `/proc` file system to determine the number of context switches for a given process. For example, the contents of the file `/proc/2166/status` will list various statistics for the process with `pid = 2166`. The command

```
cat /proc/2166/status
```

provides the following trimmed output:

```
voluntary_ctxt_switches           150
nonvoluntary_ctxt_switches        8
```

This output shows the number of context switches over the lifetime of the process. Notice the distinction between ***voluntary*** and ***nonvoluntary*** context switches. A voluntary context switch occurs when a process has given up control of the CPU because it requires a resource that is currently unavailable (such as blocking for I/O.) A nonvoluntary context switch occurs when the CPU has been taken away from a process, such as when its time slice has expired or it has been preempted by a higher-priority process.

## 5.2 Scheduling Criteria

Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.

Many criteria have been suggested for comparing CPU-scheduling algorithms. Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best. The criteria include the following:

- **CPU utilization**. We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily loaded system). (CPU utilization can be obtained by using the `top` command on Linux, macOS, and UNIX systems.)
- **Throughput**. If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed

per time unit, called **throughput**. For long processes, this rate may be one process over several seconds; for short transactions, it may be tens of processes per second.

- **Turnaround time**. From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting in the ready queue, executing on the CPU, and doing I/O.
- **Waiting time**. The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O. It affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.
- **Response time**. In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response.

It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time. In most cases, we optimize the average measure. However, under some circumstances, we prefer to optimize the minimum or maximum values rather than the average. For example, to guarantee that all users get good service, we may want to minimize the maximum response time.

Investigators have suggested that, for interactive systems (such as a PC desktop or laptop system), it is more important to minimize the variance in the response time than to minimize the average response time. A system with reasonable and predictable response time may be considered more desirable than a system that is faster on the average but is highly variable. However, little work has been done on CPU-scheduling algorithms that minimize variance.

As we discuss various CPU-scheduling algorithms in the following section, we illustrate their operation. An accurate illustration should involve many processes, each a sequence of several hundred CPU bursts and I/O bursts. For simplicity, though, we consider only one CPU burst (in milliseconds) per process in our examples. Our measure of comparison is the average waiting time. More elaborate evaluation mechanisms are discussed in Section 5.8.

## 5.3 Scheduling Algorithms

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU's core. There are many different CPU-scheduling algorithms. In this section, we describe several of them. Although most modern CPU architectures have multiple processing cores, we describe these scheduling algorithms in the context of only one processing core available. That is, a single CPU that has a single processing core, thus the system is capable of only running one process at a time. In Section 5.5 we discuss CPU scheduling in the context of multiprocessor systems.

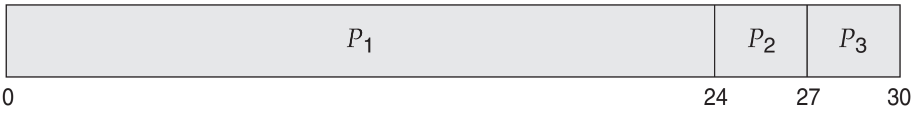
### 5.3.1 First-Come, First-Served Scheduling

By far the simplest CPU-scheduling algorithm is the **first-come first-serve** (**FCFS**) scheduling algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. The code for FCFS scheduling is simple to write and understand.

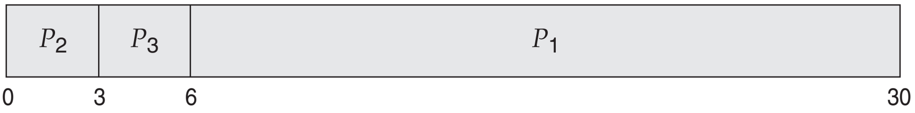
On the negative side, the average waiting time under the FCFS policy is often quite long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

| Process | Burst Time |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

If the processes arrive in the order $P_1$, $P_2$, $P_3$, and are served in FCFS order, we get the result shown in the following **Gantt chart**, which is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes:



The waiting time is 0 milliseconds for process $P_1$, 24 milliseconds for process $P_2$, and 27 milliseconds for process $P_3$. Thus, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds. If the processes arrive in the order $P_2$, $P_3$, $P_1$, however, the results will be as shown in the following Gantt chart:



The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds. This reduction is substantial. Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes' CPU burst times vary greatly.

In addition, consider the performance of FCFS scheduling in a dynamic situation. Assume we have one CPU-bound process and many I/O-bound processes. As the processes flow around the system, the following scenario may result. The CPU-bound process will get and hold the CPU. During this time, all the other processes will finish their I/O and will move into the ready queue, waiting for the CPU. While the processes wait in the ready queue, the I/O devices are idle. Eventually, the CPU-bound process finishes its CPU burst and moves to an I/O device. All the I/O-bound processes, which have short CPU bursts, execute quickly and move back to the I/O queues. At this point, the CPU sits idle. The CPU-bound process will then move back to the ready queue and be allocated the CPU. Again, all the I/O processes end up waiting in the ready queue until the CPU-bound process is done. There is a **convoy effect** as all the other processes wait for the one big process to get off the CPU. This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.

Note also that the FCFS scheduling algorithm is nonpreemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O. The FCFS algorithm is thus particularly troublesome for interactive systems, where it is important that each process get a share of the CPU at regular intervals. It would be disastrous to allow one process to keep the CPU for an extended period.

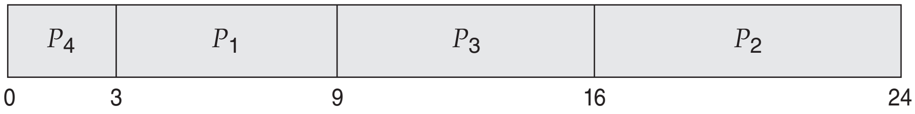
### 5.3.2 Shortest-Job-First Scheduling

A different approach to CPU scheduling is the **shortest-job-firs** (**SJF**) scheduling algorithm. This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie. Note that a more appropriate term for this scheduling method would be the ***shortest-next-CPU-burst*** algorithm, because scheduling depends on the length of the next CPU burst of a process, rather than its total length. We use the term SJF because most people and textbooks use this term to refer to this type of scheduling.

As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:

| Process | Burst Time |
|---|---|
| $P_1$ | 6 |
| $P_2$ | 8 |
| $P_3$ | 7 |
| $P_4$ | 3 |

Using SJF scheduling, we would schedule these processes according to the following Gantt chart:



The waiting time is 3 milliseconds for process $P_1$, 16 milliseconds for process $P_2$, 9 milliseconds for process $P_3$, and 0 milliseconds for process $P_4$. Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7 milliseconds. By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.

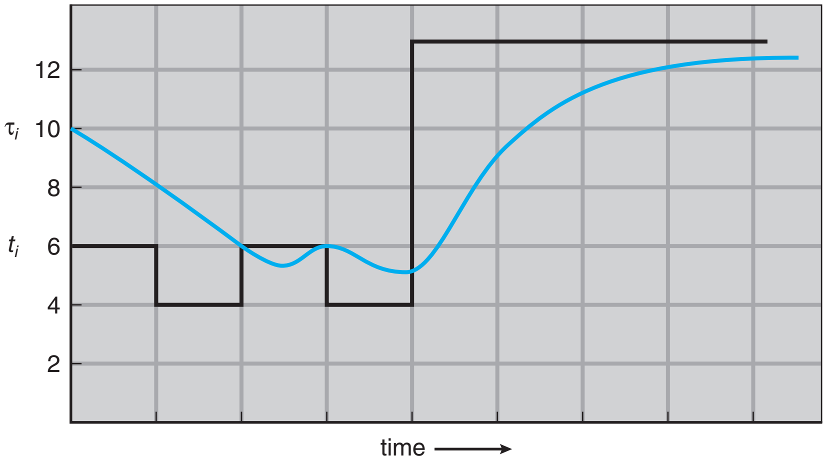
The SJF scheduling algorithm is provably optimal, in that it gives the minimum average waiting time for a given set of processes. Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the average waiting time decreases.

Although the SJF algorithm is optimal, it cannot be implemented at the level of CPU scheduling, as there is no way to know the length of the next CPU burst. One approach to this problem is to try to approximate SJF scheduling. We may not know the length of the next CPU burst, but we may be able to predict its value. We expect that the next CPU burst will be similar in length to the previous ones. By computing an approximation of the length of the next CPU burst, we can pick the process with the shortest predicted CPU burst.

The next CPU burst is generally predicted as an **exponential average** of the measured lengths of previous CPU bursts. We can define the exponential average with the following formula. Let $t_n$ be the length of the $n$th CPU burst, and let $\tau_{n+1}$ be our predicted value for the next CPU burst. Then, for $\alpha$, $0 \leq \alpha \leq 1$, define

$$\tau_{n+1} = \alpha \, t_n + (1 - \alpha)\tau_n.$$

The value of $t_n$ contains our most recent information, while $\tau_n$ stores the past history. The parameter $\alpha$ controls the relative weight of recent and past history in our prediction. If $\alpha = 0$, then $\tau_{n+1} = \tau_n$, and recent history has no effect (current conditions are assumed to be transient). If $\alpha = 1$, then $\tau_{n+1} = t_n$, and only the most recent CPU burst matters (history is assumed to be old and irrelevant). More commonly, $\alpha = 1/2$, so recent history and past history are equally weighted. The initial $\tau_0$ can be defined as a constant or as an overall system average. Figure 5.4 shows an exponential average with $\alpha = 1/2$ and $\tau_0 = 10$.



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| CPU burst ($t_i$) | | 6 | 4 | 6 | 4 | 13 | 13 | 13 | . . . |
| "guess" ($\tau_i$) | 10 | 8 | 6 | 6 | 5 | 9 | 11 | 12 | . . . |

**Figure 5.4** Prediction of the length of the next CPU burst.

To understand the behavior of the exponential average, we can expand the formula for $\tau_{n+1}$ by substituting for $\tau_n$ to find

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \cdots + (1-\alpha)^j \alpha t_{n-j} + \cdots + (1-\alpha)^{n+1}\tau_0.$$

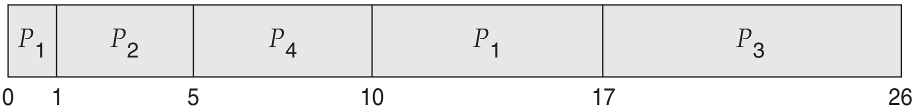
Typically, $\alpha$ is less than 1. As a result, $(1 - \alpha)$ is also less than 1, and each successive term has less weight than its predecessor.

The SJF algorithm can be either preemptive or nonpreemptive. The choice arises when a new process arrives at the ready queue while a previous process is still executing. The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process. A preemptive SJF algorithm will preempt the currently executing process, whereas a nonpreemptive SJF algorithm will allow the currently running process to finish its CPU burst. Preemptive SJF scheduling is sometimes called **shortest-remaining-time-first** scheduling.

As an example, consider the following four processes, with the length of the CPU burst given in milliseconds:

| Process | Arrival Time | Burst Time |
|---|---|---|
| $P_1$ | 0 | 8 |
| $P_2$ | 1 | 4 |
| $P_3$ | 2 | 9 |
| $P_4$ | 3 | 5 |

If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted in the following Gantt chart:



Process $P_1$ is started at time 0, since it is the only process in the queue. Process $P_2$ arrives at time 1. The remaining time for process $P_1$ (7 milliseconds) is larger than the time required by process $P_2$ (4 milliseconds), so process $P_1$ is preempted, and process $P_2$ is scheduled. The average waiting time for this example is $[(10 - 1) + (1 - 1) + (17 - 2) + (5 - 3)]/4 = 26/4 = 6.5$ milliseconds. Nonpreemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

### 5.3.3 Round-Robin Scheduling

The **round-robin** (**RR**) scheduling algorithm is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes. A small unit of time, called a **time quantum** or **time slice**, is defined. A time quantum is generally from 10 to 100 milliseconds in length. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

To implement RR scheduling, we again treat the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue.

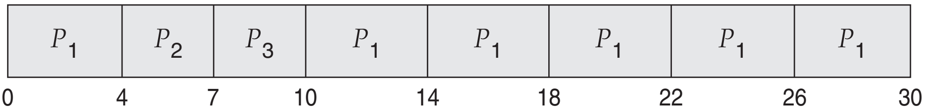
The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. If the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

| Process | Burst Time |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

If we use a time quantum of 4 milliseconds, then process $P_1$ gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process $P_2$. Process $P_2$ does not need 4 milliseconds, so it quits before its time quantum expires. The CPU is then given to the next process, process $P_3$. Once each process has received 1 time quantum, the CPU is returned to process $P_1$ for an additional time quantum. The resulting RR schedule is as follows:



Let's calculate the average waiting time for this schedule. $P_1$ waits for 6 milliseconds (10 – 4), $P_2$ waits for 4 milliseconds, and $P_3$ waits for 7 milliseconds. Thus, the average waiting time is 17/3 = 5.66 milliseconds.

In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process). If a process's CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue. The RR scheduling algorithm is thus preemptive.

If there are $n$ processes in the ready queue and the time quantum is $q$, then each process gets $1/n$ of the CPU time in chunks of at most $q$ time units. Each process must wait no longer than $(n - 1) \times q$ time units until its next time quantum. For example, with five processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds.

The performance of the RR algorithm depends heavily on the size of the time quantum. At one extreme, if the time quantum is extremely large, the RR policy is the same as the FCFS policy. In contrast, if the time quantum is extremely small (say, 1 millisecond), the RR approach can result in a large

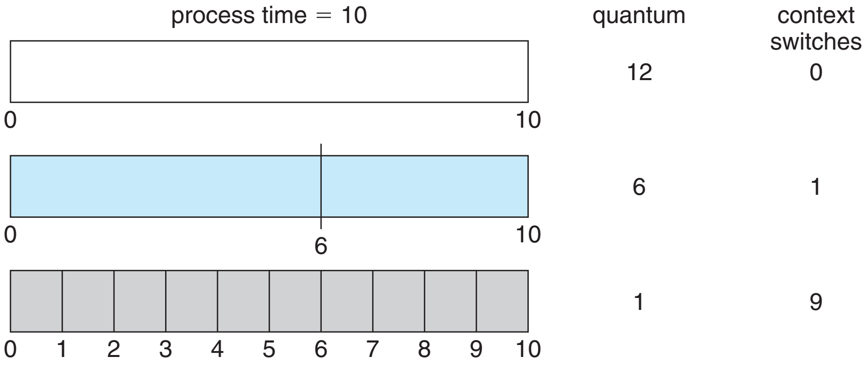


**Figure 5.5** How a smaller time quantum increases context switches.

number of context switches. Assume, for example, that we have only one process of 10 time units. If the quantum is 12 time units, the process finishes in less than 1 time quantum, with no overhead. If the quantum is 6 time units, however, the process requires 2 quanta, resulting in a context switch. If the time quantum is 1 time unit, then nine context switches will occur, slowing the execution of the process accordingly (Figure 5.5).

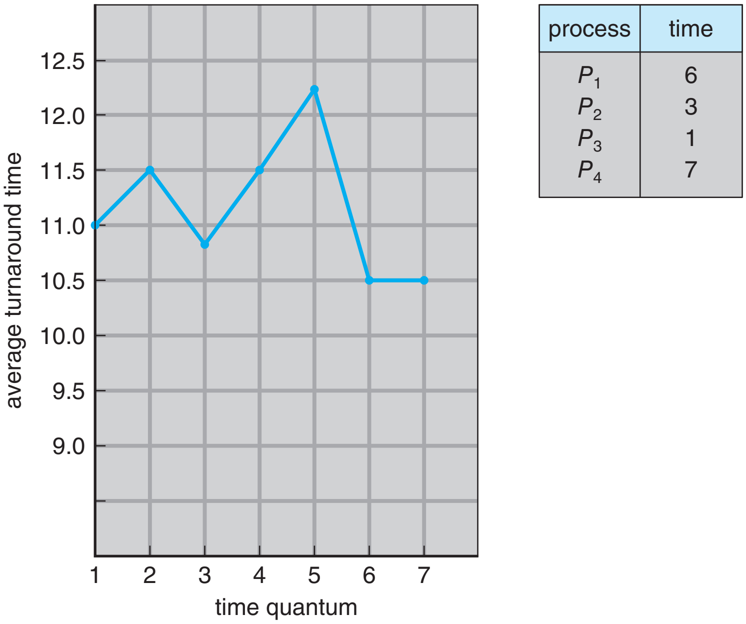
Thus, we want the time quantum to be large with respect to the context-switch time. If the context-switch time is approximately 10 percent of the time quantum, then about 10 percent of the CPU time will be spent in context switching. In practice, most modern systems have time quanta ranging from 10 to 100 milliseconds. The time required for a context switch is typically less than 10 microseconds; thus, the context-switch time is a small fraction of the time quantum.

Turnaround time also depends on the size of the time quantum. As we can see from Figure 5.6, the average turnaround time of a set of processes does not necessarily improve as the time-quantum size increases. In general, the average turnaround time can be improved if most processes finish their next CPU burst in a single time quantum. For example, given three processes of 10 time units each and a quantum of 1 time unit, the average turnaround time is 29. If the time quantum is 10, however, the average turnaround time drops to 20. If context-switch time is added in, the average turnaround time increases even more for a smaller time quantum, since more context switches are required.

Although the time quantum should be large compared with the context-switch time, it should not be too large. As we pointed out earlier, if the time quantum is too large, RR scheduling degenerates to an FCFS policy. A rule of thumb is that 80 percent of the CPU bursts should be shorter than the time quantum.

### 5.3.4 Priority Scheduling

The SJF algorithm is a special case of the general **priority-scheduling** algorithm. A priority is associated with each process, and the CPU is allocated to the



| process | time |
|---|---|
| $P_1$ | 6 |
| $P_2$ | 3 |
| $P_3$ | 1 |
| $P_4$ | 7 |

**Figure 5.6** How turnaround time varies with the time quantum.

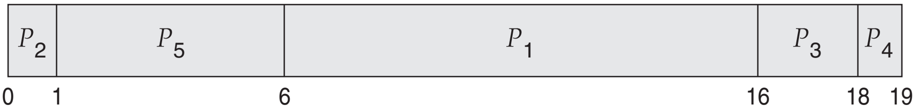
process with the highest priority. Equal-priority processes are scheduled in FCFS order. An SJF algorithm is simply a priority algorithm where the priority ($p$) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.

Note that we discuss scheduling in terms of ***high*** priority and ***low*** priority. Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095. However, there is no general agreement on whether 0 is the highest or lowest priority. Some systems use low numbers to represent low priority; others use low numbers for high priority. This difference can lead to confusion. In this text, we assume that low numbers represent high priority.

As an example, consider the following set of processes, assumed to have arrived at time 0 in the order $P_1$, $P_2$, $\cdots$, $P_5$, with the length of the CPU burst given in milliseconds:

| Process | Burst Time | Priority |
|---|---|---|
| $P_1$ | 10 | 3 |
| $P_2$ | 1 | 1 |
| $P_3$ | 2 | 4 |
| $P_4$ | 1 | 5 |
| $P_5$ | 5 | 2 |

Using priority scheduling, we would schedule these processes according to the following Gantt chart:



The average waiting time is 8.2 milliseconds.

Priorities can be defined either internally or externally. Internally defined priorities use some measurable quantity or quantities to compute the priority of a process. For example, time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used in computing priorities. External priorities are set by criteria outside the operating system, such as the importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work, and other, often political, factors.

Priority scheduling can be either preemptive or nonpreemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

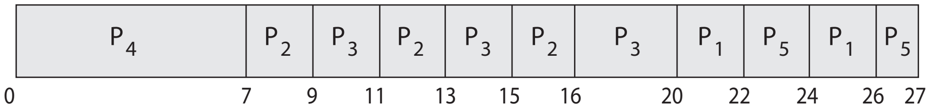
A major problem with priority scheduling algorithms is **indefinit blocking**, or **starvation**. A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algorithm can leave some low-priority processes waiting indefinitely. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU. Generally, one of two things will happen. Either the process will eventually be run (at 2 A.M. Sunday, when the system is finally lightly loaded), or the computer system will eventually crash and lose all unfinished low-priority processes. (Rumor has it that when they shut down the IBM 7094 at MIT in 1973, they found a low-priority process that had been submitted in 1967 and had not yet been run.)

A solution to the problem of indefinite blockage of low-priority processes is **aging**. Aging involves gradually increasing the priority of processes that wait in the system for a long time. For example, if priorities range from 127 (low) to 0 (high), we could periodically (say, every second) increase the priority of a waiting process by 1. Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed. In fact, it would take a little over 2 minutes for a priority-127 process to age to a priority-0 process.

Another option is to combine round-robin and priority scheduling in such a way that the system executes the highest-priority process and runs processes with the same priority using round-robin scheduling. Let's illustrate with an example using the following set of processes, with the burst time in milliseconds:

| Process | Burst Time | Priority |
|---|---|---|
| $P_1$ | 4 | 3 |
| $P_2$ | 5 | 2 |
| $P_3$ | 8 | 2 |
| $P_4$ | 7 | 1 |
| $P_5$ | 3 | 3 |

Using priority scheduling with round-robin for processes with equal priority, we would schedule these processes according to the following Gantt chart using a time quantum of 2 milliseconds:



In this example, process $P_4$ has the highest priority, so it will run to completion. Processes $P_2$ and $P_3$ have the next-highest priority, and they will execute in a round-robin fashion. Notice that when process $P_2$ finishes at time 16, process $P_3$ is the highest-priority process, so it will run until it completes execution. Now, only processes $P_1$ and $P_5$ remain, and as they have equal priority, they will execute in round-robin order until they complete.

### 5.3.5 Multilevel Queue Scheduling

With both priority and round-robin scheduling, all processes may be placed in a single queue, and the scheduler then selects the process with the highest priority to run. Depending on how the queues are managed, an $O(n)$ search may be necessary to determine the highest-priority process. In practice, it is often easier to have separate queues for each distinct priority, and priority scheduling simply schedules the process in the highest-priority queue. This is illustrated in Figure 5.7. This approach—known as **multilevel queue**—also works well when priority scheduling is combined with round-robin: if there are multiple processes in the highest-priority queue, they are executed in round-robin order. In the most generalized form of this approach, a priority is assigned statically to each process, and a process remains in the same queue for the duration of its runtime.

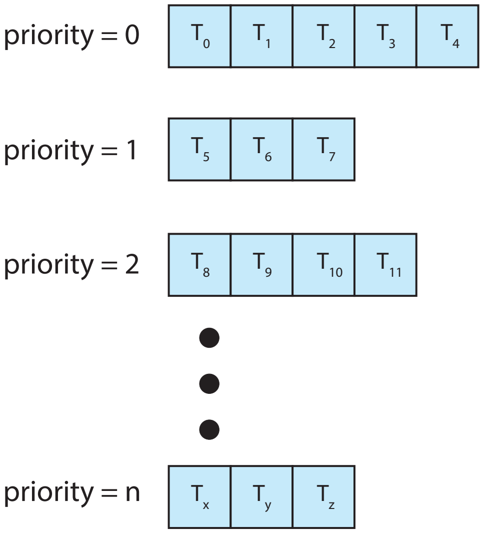


**Figure 5.7** Separate queues for each priority.

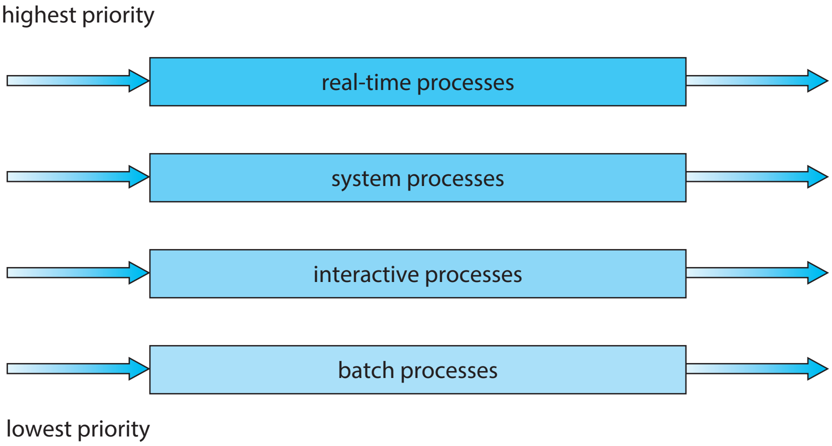


**Figure 5.8** Multilevel queue scheduling.

A multilevel queue scheduling algorithm can also be used to partition processes into several separate queues based on the process type (Figure 5.8). For example, a common division is made between **foreground** (interactive) processes and **background** (batch) processes. These two types of processes have different response-time requirements and so may have different scheduling needs. In addition, foreground processes may have priority (externally defined) over background processes. Separate queues might be used for foreground and background processes, and each queue might have its own scheduling algorithm. The foreground queue might be scheduled by an RR algorithm, for example, while the background queue is scheduled by an FCFS algorithm.

In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling. For example, the real-time queue may have absolute priority over the interactive queue.

Let's look at an example of a multilevel queue scheduling algorithm with four queues, listed below in order of priority:

1. Real-time processes
2. System processes
3. Interactive processes
4. Batch processes

Each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, could run unless the queues for real-time processes, system processes, and interactive processes were all empty. If an interactive process entered the ready queue while a batch process was running, the batch process would be preempted.

Another possibility is to time-slice among the queues. Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes. For instance, in the foreground–background queue example, the foreground queue can be given 80 percent of the CPU time for RR scheduling among its processes, while the background queue receives 20 percent of the CPU to give to its processes on an FCFS basis.

### 5.3.6 Multilevel Feedback Queue Scheduling

Normally, when the multilevel queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system. If there are separate queues for foreground and background processes, for example, processes do not move from one queue to the other, since processes do not change their foreground or background nature. This setup has the advantage of low scheduling overhead, but it is inflexible.

The **multilevel feedback queue** scheduling algorithm, in contrast, allows a process to move between queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes—which are typically characterized by short CPU bursts—in the higher-priority queues. In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.

For example, consider a multilevel feedback queue scheduler with three queues, numbered from 0 to 2 (Figure 5.9). The scheduler first executes all processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1. Similarly, processes in queue 2 will be executed only if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2. A process in queue 1 will in turn be preempted by a process arriving for queue 0.

An entering process is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds. If it does not complete, it is preempted and is put into queue 2. Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty. To prevent starvation, a process that waits too long in a lower-priority queue may gradually be moved to a higher-priority queue.

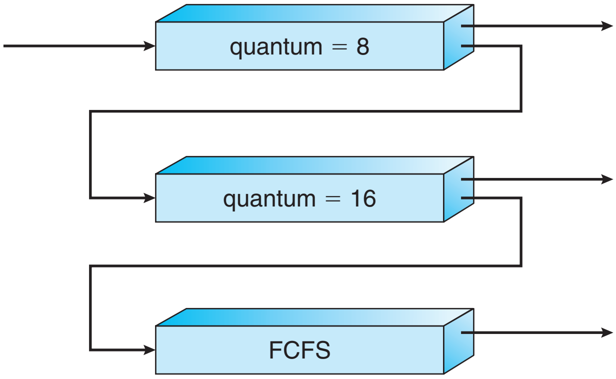


**Figure 5.9** Multilevel feedback queues.

This scheduling algorithm gives highest priority to any process with a CPU burst of 8 milliseconds or less. Such a process will quickly get the CPU, finish its CPU burst, and go off to its next I/O burst. Processes that need more than 8 but less than 24 milliseconds are also served quickly, although with lower priority than shorter processes. Long processes automatically sink to queue 2 and are served in FCFS order with any CPU cycles left over from queues 0 and 1.

In general, a multilevel feedback queue scheduler is defined by the following parameters:

- The number of queues
- The scheduling algorithm for each queue
- The method used to determine when to upgrade a process to a higher-priority queue
- The method used to determine when to demote a process to a lower-priority queue
- The method used to determine which queue a process will enter when that process needs service

The definition of a multilevel feedback queue scheduler makes it the most general CPU-scheduling algorithm. It can be configured to match a specific system under design. Unfortunately, it is also the most complex algorithm, since defining the best scheduler requires some means by which to select values for all the parameters.

## 5.4 Thread Scheduling

In Chapter 4, we introduced threads to the process model, distinguishing between ***user-level*** and ***kernel-level*** threads. On most modern operating systems it is kernel-level threads—not processes—that are being scheduled by the operating system. User-level threads are managed by a thread library, and the kernel is unaware of them. To run on a CPU, user-level threads must ultimately be mapped to an associated kernel-level thread, although this mapping may be indirect and may use a lightweight process (LWP). In this section, we explore scheduling issues involving user-level and kernel-level threads and offer specific examples of scheduling for Pthreads.

### 5.4.1 Contention Scope

One distinction between user-level and kernel-level threads lies in how they are scheduled. On systems implementing the many-to-one (Section 4.3.1) and many-to-many (Section 4.3.3) models, the thread library schedules user-level threads to run on an available LWP. This scheme is known as **process-contention scope** (**PCS**), since competition for the CPU takes place among threads belonging to the same process. (When we say the thread library *schedules* user threads onto available LWPs, we do not mean that the threads are actually *running* on a CPU as that further requires the operating system to schedule the LWP's kernel thread onto a physical CPU core.) To decide which kernel-level thread to schedule onto a CPU, the kernel uses **system-contention scope** (**SCS**). Competition for the CPU with SCS scheduling takes place among all threads in the system. Systems using the one-to-one model (Section 4.3.2), such as Windows and Linux schedule threads using only SCS.

Typically, PCS is done according to priority—the scheduler selects the runnable thread with the highest priority to run. User-level thread priorities are set by the programmer and are not adjusted by the thread library, although some thread libraries may allow the programmer to change the priority of a thread. It is important to note that PCS will typically preempt the thread currently running in favor of a higher-priority thread; however, there is no guarantee of time slicing (Section 5.3.3) among threads of equal priority.

### 5.4.2 Pthread Scheduling

We provided a sample POSIX Pthread program in Section 4.4.1, along with an introduction to thread creation with Pthreads. Now, we highlight the POSIX Pthread API that allows specifying PCS or SCS during thread creation. Pthreads identifies the following contention scope values:

- PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling.
- PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling.

On systems implementing the many-to-many model, the PTHREAD_SCOPE_PROCESS policy schedules user-level threads onto available LWPs. The number of LWPs is maintained by the thread library, perhaps using scheduler activations (Section 4.6.5). The PTHREAD_SCOPE_SYSTEM scheduling policy will create and bind an LWP for each user-level thread on many-to-many systems, effectively mapping threads using the one-to-one policy.

The Pthread IPC (Interprocess Communication) provides two functions for setting—and getting—the contention scope policy:

- `pthread_attr_setscope(pthread_attr_t *attr, int scope)`
- `pthread_attr_getscope(pthread_attr_t *attr, int *scope)`

The first parameter for both functions contains a pointer to the attribute set for the thread. The second parameter for the `pthread_attr_setscope()` function is passed either the PTHREAD_SCOPE_SYSTEM or the PTHREAD_SCOPE_PROCESS value, indicating how the contention scope is to be set. In the case of `pthread_attr_getscope()`, this second parameter contains a pointer to an `int` value that is set to the current value of the contention scope. If an error occurs, each of these functions returns a nonzero value.

In Figure 5.10, we illustrate a Pthread scheduling API. The program first determines the existing contention scope and sets it to PTHREAD_SCOPE_SYSTEM. It then creates five separate threads that will run using the SCS scheduling policy. Note that on some systems, only certain contention scope values are allowed. For example, Linux and macOS systems allow only PTHREAD_SCOPE_SYSTEM.

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char *argv[])
{
   int i, scope;
   pthread_t tid[NUM_THREADS];
   pthread_attr_t attr;

   /* get the default attributes */
   pthread_attr_init(&attr);

   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
      fprintf(stderr, "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
       printf("PTHREAD_SCOPE_PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
       printf("PTHREAD_SCOPE_SYSTEM");
      else
       fprintf(stderr, "Illegal scope value.\n");
    }

   /* set the scheduling algorithm to PCS or SCS */
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
       pthread_create(&tid[i],&attr,runner,NULL);

   /* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
       pthread_join(tid[i], NULL);
}

/* Each thread will begin control in this function */
void *runner(void *param)
{
   /* do some work ... */

   pthread_exit(0);
}
```

**Figure 5.10** Pthread scheduling API.

## 5.5 Multi-Processor Scheduling

Our discussion thus far has focused on the problems of scheduling the CPU in a system with a single processing core. If multiple CPUs are available, **load sharing**, where multiple threads may run in parallel, becomes possible, however scheduling issues become correspondingly more complex. Many possibilities have been tried; and as we saw with CPU scheduling with a single-core CPU, there is no one best solution.

Traditionally, the term **multiprocessor** referred to systems that provided multiple physical processors, where each processor contained one single-core CPU. However, the definition of multiprocessor has evolved significantly, and on modern computing systems, ***multiprocessor*** now applies to the following system architectures:

- Multicore CPUs
- Multithreaded cores
- NUMA systems
- Heterogeneous multiprocessing

Here, we discuss several concerns in multiprocessor scheduling in the context of these different architectures. In the first three examples we concentrate on systems in which the processors are identical—homogeneous—in terms of their functionality. We can then use any available CPU to run any process in the queue. In the last example we explore a system where the processors are not identical in their capabilities.

### 5.5.1 Approaches to Multiple-Processor Scheduling

One approach to CPU scheduling in a multiprocessor system has all scheduling decisions, I/O processing, and other system activities handled by a single processor — the master server. The other processors execute only user code. This **asymmetric multiprocessing** is simple because only one core accesses the system data structures, reducing the need for data sharing. The downfall of this approach is the master server becomes a potential bottleneck where overall system performance may be reduced.

The standard approach for supporting multiprocessors is **symmetric multiprocessing** (**SMP**), where each processor is self-scheduling. Scheduling proceeds by having the scheduler for each processor examine the ready queue and select a thread to run. Note that this provides two possible strategies for organizing the threads eligible to be scheduled:

1. All threads may be in a common ready queue.
2. Each processor may have its own private queue of threads.

These two strategies are contrasted in Figure 5.11. If we select the first option, we have a possible race condition on the shared ready queue and therefore must ensure that two separate processors do not choose to schedule the same thread and that threads are not lost from the queue. As discussed in

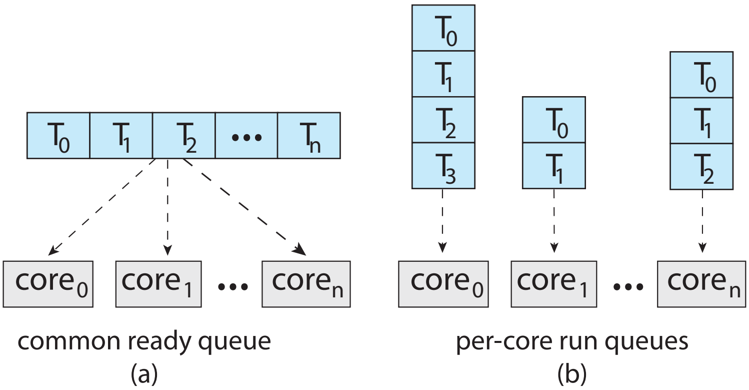


**Figure 5.11** Organization of ready queues.

Chapter 6, we could use some form of locking to protect the common ready queue from this race condition. Locking would be highly contended, however, as all accesses to the queue would require lock ownership, and accessing the shared queue would likely be a performance bottleneck. The second option permits each processor to schedule threads from its private run queue and therefore does not suffer from the possible performance problems associated with a shared run queue. Thus, it is the most common approach on systems supporting SMP. Additionally, as described in Section 5.5.4, having private, per-processor run queues in fact may lead to more efficient use of cache memory. There are issues with per-processor run queues—most notably, workloads of varying sizes. However, as we shall see, balancing algorithms can be used to equalize workloads among all processors.

Virtually all modern operating systems support SMP, including Windows, Linux, and macOS as well as mobile systems including Android and iOS. In the remainder of this section, we discuss issues concerning SMP systems when designing CPU scheduling algorithms.

### 5.5.2 Multicore Processors

Traditionally, SMP systems have allowed several processes to run in parallel by providing multiple physical processors. However, most contemporary computer hardware now places multiple computing cores on the same physical chip, resulting in a **multicore processor**. Each core maintains its architectural state and thus appears to the operating system to be a separate logical CPU. SMP systems that use multicore processors are faster and consume less power than systems in which each CPU has its own physical chip.

Multicore processors may complicate scheduling issues. Let's consider how this can happen. Researchers have discovered that when a processor accesses memory, it spends a significant amount of time waiting for the data to become available. This situation, known as a **memory stall**, occurs primarily because modern processors operate at much faster speeds than memory. However, a memory stall can also occur because of a cache miss (accessing data that are not in cache memory). Figure 5.12 illustrates a memory stall. In this scenario, the processor can spend up to 50 percent of its time waiting for data to become available from memory.

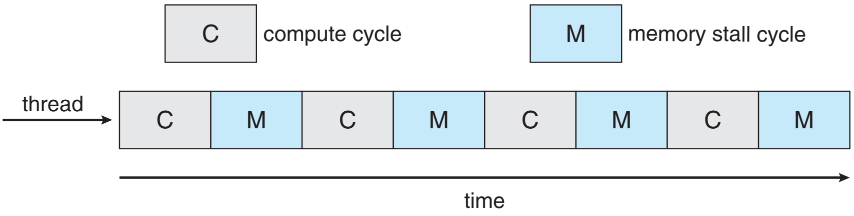


**Figure 5.12** Memory stall.

To remedy this situation, many recent hardware designs have implemented multithreaded processing cores in which two (or more) **hardware threads** are assigned to each core. That way, if one hardware thread stalls while waiting for memory, the core can switch to another thread. Figure 5.13 illustrates a dual-threaded processing core on which the execution of thread 0 and the execution of thread 1 are interleaved. From an operating system perspective, each hardware thread maintains its architectural state, such as instruction pointer and register set, and thus appears as a logical CPU that is available to run a software thread. This technique—known as **chip multithreading** (CMT) —is illustrated in Figure 5.14. Here, the processor contains four computing cores, with each core containing two hardware threads. From the perspective of the operating system, there are eight logical CPUs.

Intel processors use the term **hyper-threading** (also known as **simultaneous multithreading** or SMT) to describe assigning multiple hardware threads to a single processing core. Contemporary Intel processors—such as the i7—support two threads per core, while the Oracle Sparc M7 processor supports eight threads per core, with eight cores per processor, thus providing the operating system with 64 logical CPUs.

In general, there are two ways to multithread a processing core: **coarse-grained** and **fine-grained** multithreading. With coarse-grained multithreading, a thread executes on a core until a long-latency event such as a memory stall occurs. Because of the delay caused by the long-latency event, the core must switch to another thread to begin execution. However, the cost of switching between threads is high, since the instruction pipeline must be flushed before the other thread can begin execution on the processor core. Once this new thread begins execution, it begins filling the pipeline with its instructions. Fine-grained (or interleaved) multithreading switches between threads at a much finer level of granularity—typically at the boundary of an instruction

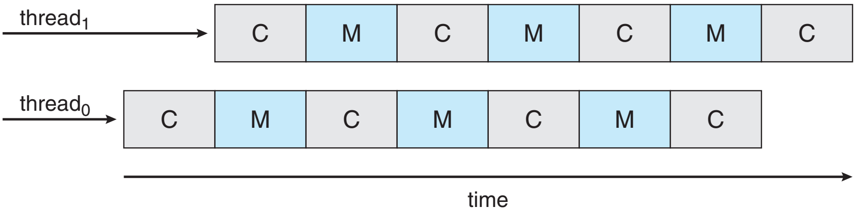


**Figure 5.13** Multithreaded multicore system.

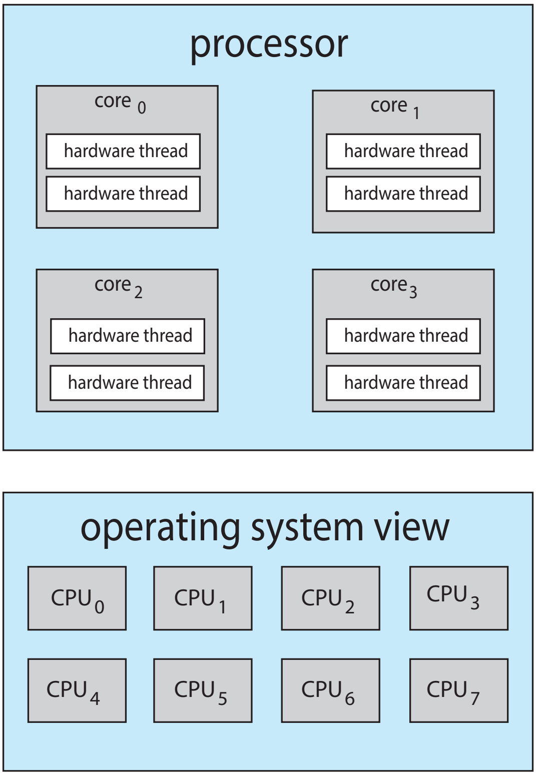

processor
core 0
hardware thread
hardware thread
core 1
hardware thread
hardware thread
core 2
hardware thread
hardware thread
core 3
hardware thread
hardware thread
operating system view
CPU 0
CPU 1
CPU 2
CPU 3
CPU 4
CPU 5
CPU 6
CPU 7

**Figure 5.14** Chip multithreading.

cycle. However, the architectural design of fine-grained systems includes logic for thread switching. As a result, the cost of switching between threads is small.

It is important to note that the resources of the physical core (such as caches and pipelines) must be shared among its hardware threads, and therefore a processing core can only execute one hardware thread at a time. Consequently, a multithreaded, multicore processor actually requires two different levels of scheduling, as shown in Figure 5.15, which illustrates a dual-threaded processing core.

On one level are the scheduling decisions that must be made by the operating system as it chooses which software thread to run on each hardware thread (logical CPU). For all practical purposes, such decisions have been the primary focus of this chapter. Therefore, for this level of scheduling, the operating system may choose any scheduling algorithm, including those described in Section 5.3.

A second level of scheduling specifies how each core decides which hardware thread to run. There are several strategies to adopt in this situation. One approach is to use a simple round-robin algorithm to schedule a hardware thread to the processing core. This is the approach adopted by the UltraSPARC T3. Another approach is used by the Intel Itanium, a dual-core processor with two hardware-managed threads per core. Assigned to each hardware thread is a dynamic ***urgency*** value ranging from 0 to 7, with 0 representing the lowest urgency and 7 the highest. The Itanium identifies five different events that may

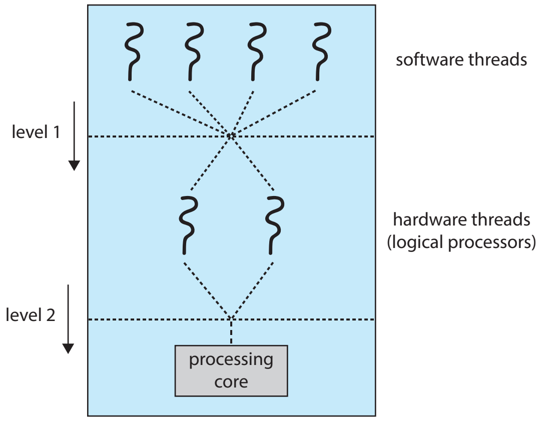


**Figure 5.15** Two levels of scheduling.

trigger a thread switch. When one of these events occurs, the thread-switching logic compares the urgency of the two threads and selects the thread with the highest urgency value to execute on the processor core.

Note that the two different levels of scheduling shown in Figure 5.15 are not necessarily mutually exclusive. In fact, if the operating system scheduler (the first level) is made aware of the sharing of processor resources, it can make more effective scheduling decisions. As an example, assume that a CPU has two processing cores, and each core has two hardware threads. If two software threads are running on this system, they can be running either on the same core or on separate cores. If they are both scheduled to run on the same core, they have to share processor resources and thus are likely to proceed more slowly than if they were scheduled on separate cores. If the operating system is aware of the level of processor resource sharing, it can schedule software threads onto logical processors that do not share resources.

### 5.5.3 Load Balancing

On SMP systems, it is important to keep the workload balanced among all processors to fully utilize the benefits of having more than one processor. Otherwise, one or more processors may sit idle while other processors have high workloads, along with ready queues of threads awaiting the CPU. **Load balancing** attempts to keep the workload evenly distributed across all processors in an SMP system. It is important to note that load balancing is typically necessary only on systems where each processor has its own private ready queue of eligible threads to execute. On systems with a common run queue, load balancing is unnecessary, because once a processor becomes idle, it immediately extracts a runnable thread from the common ready queue.

There are two general approaches to load balancing: push migration and pull migration. With **push migration**, a specific task periodically checks the load on each processor and—if it finds an imbalance—evenly distributes the load by moving (or pushing) threads from overloaded to idle or less-busy processors. **Pull migration** occurs when an idle processor pulls a waiting task from a busy processor. Push and pull migration need not be mutually exclusive and are, in fact, often implemented in parallel on load-balancing systems. For example, the Linux CFS scheduler (described in Section 5.7.1) and the ULE scheduler available for FreeBSD systems implement both techniques.

The concept of a “balanced load” may have different meanings. One view of a balanced load may require simply that all queues have approximately the same number of threads. Alternatively, balance may require an equal distribution of thread priorities across all queues. In addition, in certain situations, neither of these strategies may be sufficient. Indeed, they may work against the goals of the scheduling algorithm. (We leave further consideration of this as an exercise.)

### 5.5.4 Processor Affinity

Consider what happens to cache memory when a thread has been running on a specific processor. The data most recently accessed by the thread populate the cache for the processor. As a result, successive memory accesses by the thread are often satisfied in cache memory (known as a “warm cache”). Now consider what happens if the thread migrates to another processor—say, due to load balancing. The contents of cache memory must be invalidated for the first processor, and the cache for the second processor must be repopulated. Because of the high cost of invalidating and repopulating caches, most operating systems with SMP support try to avoid migrating a thread from one processor to another and instead attempt to keep a thread running on the same processor and take advantage of a warm cache. This is known as **processor affinity**—that is, a process has an affinity for the processor on which it is currently running.

The two strategies described in Section 5.5.1 for organizing the queue of threads available for scheduling have implications for processor affinity. If we adopt the approach of a common ready queue, a thread may be selected for execution by any processor. Thus, if a thread is scheduled on a new processor, that processor’s cache must be repopulated. With private, per-processor ready queues, a thread is always scheduled on the same processor and can therefore benefit from the contents of a warm cache. Essentially, per-processor ready queues provide processor affinity for free!

Processor affinity takes several forms. When an operating system has a policy of attempting to keep a process running on the same processor—but not guaranteeing that it will do so—we have a situation known as **soft affinity**. Here, the operating system will attempt to keep a process on a single processor, but it is possible for a process to migrate between processors during load balancing. In contrast, some systems provide system calls that support **hard affinity**, thereby allowing a process to specify a subset of processors on which it can run. Many systems provide both soft and hard affinity. For example, Linux implements soft affinity, but it also provides the `sched_setaffinity()` system call, which supports hard affinity by allowing a thread to specify the set of CPUs on which it is eligible to run.

The main-memory architecture of a system can affect processor affinity issues as well. Figure 5.16 illustrates an architecture featuring non-uniform memory access (NUMA) where there are two physical processor chips each with their own CPU and local memory. Although a system interconnect allows all CPUs in a NUMA system to share one physical address space, a CPU has faster access to its local memory than to memory local to another CPU. If the operating system’s CPU scheduler and memory-placement algorithms are ***NUMA-aware***

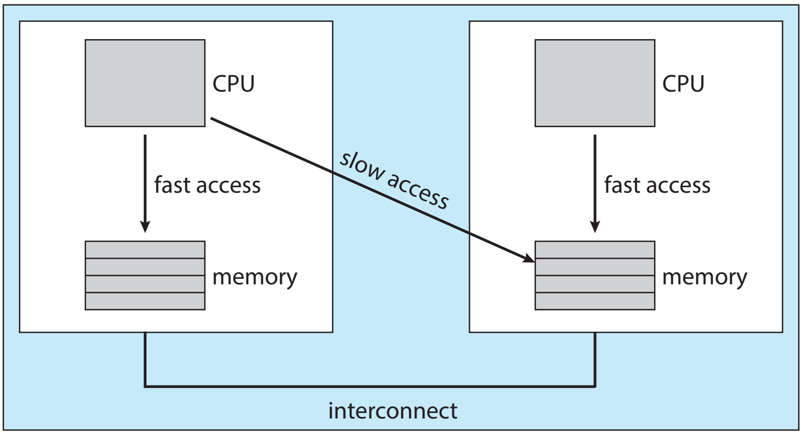


**Figure 5.16** NUMA and CPU scheduling.

and work together, then a thread that has been scheduled onto a particular CPU can be allocated memory closest to where the CPU resides, thus providing the thread the fastest possible memory access.

Interestingly, load balancing often counteracts the benefits of processor affinity. That is, the benefit of keeping a thread running on the same processor is that the thread can take advantage of its data being in that processor's cache memory. Balancing loads by moving a thread from one processor to another removes this benefit. Similarly, migrating a thread between processors may incur a penalty on NUMA systems, where a thread may be moved to a processor that requires longer memory access times. In other words, there is a natural tension between load balancing and minimizing memory access times. Thus, scheduling algorithms for modern multicore NUMA systems have become quite complex. In Section 5.7.1, we examine the Linux CFS scheduling algorithm and explore how it balances these competing goals.

### 5.5.5 Heterogeneous Multiprocessing

In the examples we have discussed so far, all processors are identical in terms of their capabilities, thus allowing any thread to run on any processing core. The only difference being that memory access times may vary based upon load balancing and processor affinity policies, as well as on NUMA systems.

Although mobile systems now include multicore architectures, some systems are now designed using cores that run the same instruction set, yet vary in terms of their clock speed and power management, including the ability to adjust the power consumption of a core to the point of idling the core. Such systems are known as **heterogeneous multiprocessing** (HMP). Note this is not a form of asymmetric multiprocessing as described in Section 5.5.1 as both system and user tasks can run on any core. Rather, the intention behind HMP is to better manage power consumption by assigning tasks to certain cores based upon the specific demands of the task.

For ARM processors that support it, this type of architecture is known as **big.LITTLE** where higher-peformance ***big*** cores are combined with energy efficient ***LITTLE*** cores. ***Big*** cores consume greater energy and therefore should only be used for short periods of time. Likewise, *little* cores use less energy and can therefore be used for longer periods.

There are several advantages to this approach. By combining a number of slower cores with faster ones, a CPU scheduler can assign tasks that do not require high performance, but may need to run for longer periods, (such as background tasks) to little cores, thereby helping to preserve a battery charge. Similarly, interactive applications which require more processing power, but may run for shorter durations, can be assigned to big cores. Additionally, if the mobile device is in a power-saving mode, energy-intensive big cores can be disabled and the system can rely solely on energy-efficient little cores. Windows 10 supports HMP scheduling by allowing a thread to select a scheduling policy that best supports its power management demands.

## 5.6 Real-Time CPU Scheduling

CPU scheduling for real-time operating systems involves special issues. In general, we can distinguish between soft real-time systems and hard real-time systems. **Soft real-time systems** provide no guarantee as to when a critical real-time process will be scheduled. They guarantee only that the process will be given preference over noncritical processes. **Hard real-time systems** have stricter requirements. A task must be serviced by its deadline; service after the deadline has expired is the same as no service at all. In this section, we explore several issues related to process scheduling in both soft and hard real-time operating systems.

### 5.6.1 Minimizing Latency

Consider the event-driven nature of a real-time system. The system is typically waiting for an event in real time to occur. Events may arise either in software —as when a timer expires—or in hardware—as when a remote-controlled vehicle detects that it is approaching an obstruction. When an event occurs, the system must respond to and service it as quickly as possible. We refer to **event latency** as the amount of time that elapses from when an event occurs to when it is serviced (Figure 5.17).

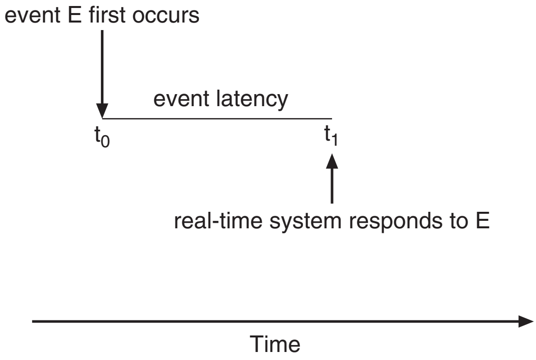


**Figure 5.17** Event latency.

Usually, different events have different latency requirements. For example, the latency requirement for an antilock brake system might be 3 to 5 milliseconds. That is, from the time a wheel first detects that it is sliding, the system controlling the antilock brakes has 3 to 5 milliseconds to respond to and control the situation. Any response that takes longer might result in the automobile's veering out of control. In contrast, an embedded system controlling radar in an airliner might tolerate a latency period of several seconds.

Two types of latencies affect the performance of real-time systems:

1. Interrupt latency
2. Dispatch latency

**Interrupt latency** refers to the period of time from the arrival of an interrupt at the CPU to the start of the routine that services the interrupt. When an interrupt occurs, the operating system must first complete the instruction it is executing and determine the type of interrupt that occurred. It must then save the state of the current process before servicing the interrupt using the specific interrupt service routine (ISR). The total time required to perform these tasks is the interrupt latency (Figure 5.18).

Obviously, it is crucial for real-time operating systems to minimize interrupt latency to ensure that real-time tasks receive immediate attention. Indeed, for hard real-time systems, interrupt latency must not simply be minimized, it must be bounded to meet the strict requirements of these systems.

One important factor contributing to interrupt latency is the amount of time interrupts may be disabled while kernel data structures are being updated. Real-time operating systems require that interrupts be disabled for only very short periods of time.

The amount of time required for the scheduling dispatcher to stop one process and start another is known as **dispatch latency**. Providing real-time

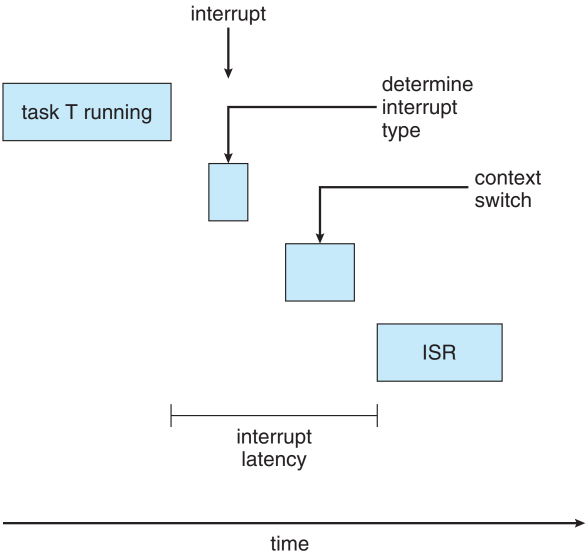


**Figure 5.18** Interrupt latency.

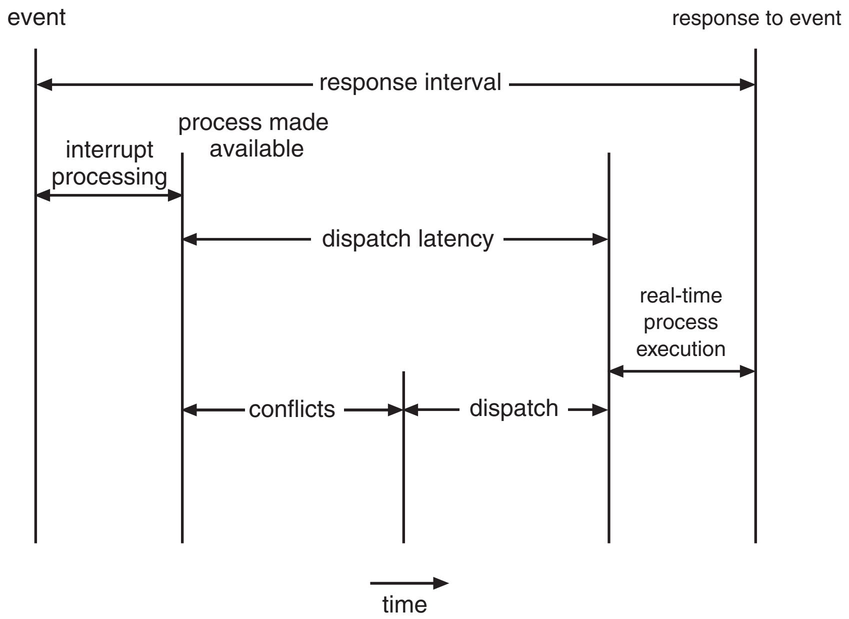


**Figure 5.19** Dispatch latency.

tasks with immediate access to the CPU mandates that real-time operating systems minimize this latency as well. The most effective technique for keeping dispatch latency low is to provide preemptive kernels. For hard real-time systems, dispatch latency is typically measured in several microseconds.

In Figure 5.19, we diagram the makeup of dispatch latency. The **conflict phase** of dispatch latency has two components:

1. Preemption of any process running in the kernel
2. Release by low-priority processes of resources needed by a high-priority process

Following the conflict phase, the dispatch phase schedules the high-priority process onto an available CPU.

### 5.6.2 Priority-Based Scheduling

The most important feature of a real-time operating system is to respond immediately to a real-time process as soon as that process requires the CPU. As a result, the scheduler for a real-time operating system must support a priority-based algorithm with preemption. Recall that priority-based scheduling algorithms assign each process a priority based on its importance; more important tasks are assigned higher priorities than those deemed less important. If the scheduler also supports preemption, a process currently running on the CPU will be preempted if a higher-priority process becomes available to run.

Preemptive, priority-based scheduling algorithms are discussed in detail in Section 5.3.4, and Section 5.7 presents examples of the soft real-time scheduling features of the Linux, Windows, and Solaris operating systems. Each of these systems assigns real-time processes the highest scheduling priority. For

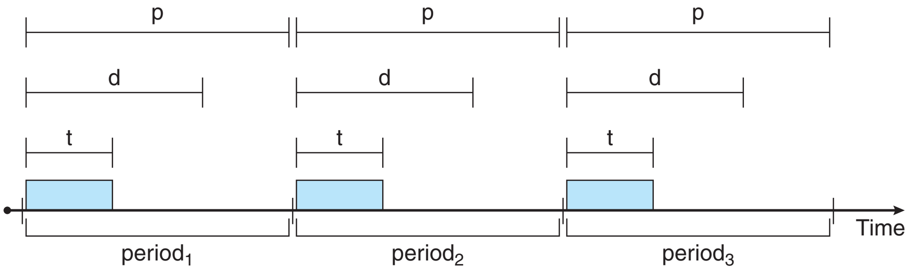


**Figure 5.20** Periodic task.

example, Windows has 32 different priority levels. The highest levels—priority values 16 to 31—are reserved for real-time processes. Solaris and Linux have similar prioritization schemes.

Note that providing a preemptive, priority-based scheduler only guarantees soft real-time functionality. Hard real-time systems must further guarantee that real-time tasks will be serviced in accord with their deadline requirements, and making such guarantees requires additional scheduling features. In the remainder of this section, we cover scheduling algorithms appropriate for hard real-time systems.

Before we proceed with the details of the individual schedulers, however, we must define certain characteristics of the processes that are to be scheduled. First, the processes are considered **periodic**. That is, they require the CPU at constant intervals (periods). Once a periodic process has acquired the CPU, it has a fixed processing time $t$, a deadline $d$ by which it must be serviced by the CPU, and a period $p$. The relationship of the processing time, the deadline, and the period can be expressed as $0 \leq t \leq d \leq p$. The **rate** of a periodic task is $1/p$. Figure 5.20 illustrates the execution of a periodic process over time. Schedulers can take advantage of these characteristics and assign priorities according to a process's deadline or rate requirements.

What is unusual about this form of scheduling is that a process may have to announce its deadline requirements to the scheduler. Then, using a technique known as an **admission-control** algorithm, the scheduler does one of two things. It either admits the process, guaranteeing that the process will complete on time, or rejects the request as impossible if it cannot guarantee that the task will be serviced by its deadline.

### 5.6.3 Rate-Monotonic Scheduling

The **rate-monotonic** scheduling algorithm schedules periodic tasks using a static priority policy with preemption. If a lower-priority process is running and a higher-priority process becomes available to run, it will preempt the lower-priority process. Upon entering the system, each periodic task is assigned a priority inversely based on its period. The shorter the period, the higher the priority; the longer the period, the lower the priority. The rationale behind this policy is to assign a higher priority to tasks that require the CPU more often. Furthermore, rate-monotonic scheduling assumes that the process-

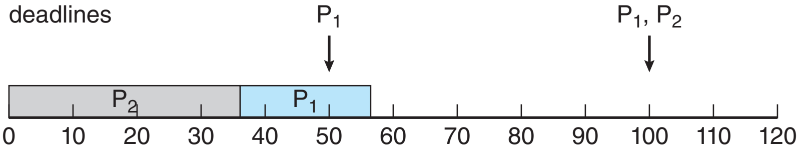


**Figure 5.21** Scheduling of tasks when $P_2$ has a higher priority than $P_1$.

ing time of a periodic process is the same for each CPU burst. That is, every time a process acquires the CPU, the duration of its CPU burst is the same.

Let's consider an example. We have two processes, $P_1$ and $P_2$. The periods for $P_1$ and $P_2$ are 50 and 100, respectively—that is, $p_1 = 50$ and $p_2 = 100$. The processing times are $t_1 = 20$ for $P_1$ and $t_2 = 35$ for $P_2$. The deadline for each process requires that it complete its CPU burst by the start of its next period.

We must first ask ourselves whether it is possible to schedule these tasks so that each meets its deadlines. If we measure the CPU utilization of a process $P_i$ as the ratio of its burst to its period—$t_i/p_i$—the CPU utilization of $P_1$ is $20/50 = 0.40$ and that of $P_2$ is $35/100 = 0.35$, for a total CPU utilization of 75 percent. Therefore, it seems we can schedule these tasks in such a way that both meet their deadlines and still leave the CPU with available cycles.

Suppose we assign $P_2$ a higher priority than $P_1$. The execution of $P_1$ and $P_2$ in this situation is shown in Figure 5.21. As we can see, $P_2$ starts execution first and completes at time 35. At this point, $P_1$ starts; it completes its CPU burst at time 55. However, the first deadline for $P_1$ was at time 50, so the scheduler has caused $P_1$ to miss its deadline.

Now suppose we use rate-monotonic scheduling, in which we assign $P_1$ a higher priority than $P_2$ because the period of $P_1$ is shorter than that of $P_2$. The execution of these processes in this situation is shown in Figure 5.22. $P_1$ starts first and completes its CPU burst at time 20, thereby meeting its first deadline. $P_2$ starts running at this point and runs until time 50. At this time, it is preempted by $P_1$, although it still has 5 milliseconds remaining in its CPU burst. $P_1$ completes its CPU burst at time 70, at which point the scheduler resumes $P_2$. $P_2$ completes its CPU burst at time 75, also meeting its first deadline. The system is idle until time 100, when $P_1$ is scheduled again.

Rate-monotonic scheduling is considered optimal in that if a set of processes cannot be scheduled by this algorithm, it cannot be scheduled by any other algorithm that assigns static priorities. Let's next examine a set of processes that cannot be scheduled using the rate-monotonic algorithm.

Assume that process $P_1$ has a period of $p_1 = 50$ and a CPU burst of $t_1 = 25$. For $P_2$, the corresponding values are $p_2 = 80$ and $t_2 = 35$. Rate-monotonic

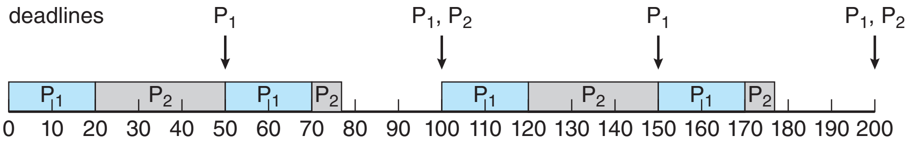


**Figure 5.22** Rate-monotonic scheduling.

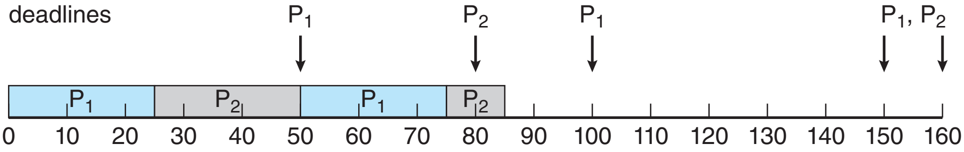


**Figure 5.23** Missing deadlines with rate-monotonic scheduling.

scheduling would assign process $P_1$ a higher priority, as it has the shorter period. The total CPU utilization of the two processes is (25/50) + (35/80) = 0.94, and it therefore seems logical that the two processes could be scheduled and still leave the CPU with 6 percent available time. Figure 5.23 shows the scheduling of processes $P_1$ and $P_2$. Initially, $P_1$ runs until it completes its CPU burst at time 25. Process $P_2$ then begins running and runs until time 50, when it is preempted by $P_1$. At this point, $P_2$ still has 10 milliseconds remaining in its CPU burst. Process $P_1$ runs until time 75; consequently, P2 finishes its burst at time 85, after the deadline for completion of its CPU burst at time 80.

Despite being optimal, then, rate-monotonic scheduling has a limitation: CPU utilization is bounded, and it is not always possible to maximize CPU resources fully. The worst-case CPU utilization for scheduling $N$ processes is

$$N(2^{1/N} - 1).$$

With one process in the system, CPU utilization is 100 percent, but it falls to approximately 69 percent as the number of processes approaches infinity. With two processes, CPU utilization is bounded at about 83 percent. Combined CPU utilization for the two processes scheduled in Figure 5.21 and Figure 5.22 is 75 percent; therefore, the rate-monotonic scheduling algorithm is guaranteed to schedule them so that they can meet their deadlines. For the two processes scheduled in Figure 5.23, combined CPU utilization is approximately 94 percent; therefore, rate-monotonic scheduling cannot guarantee that they can be scheduled so that they meet their deadlines.

### 5.6.4 Earliest-Deadline-First Scheduling

**Earliest-deadline-firs** (**EDF**) scheduling assigns priorities dynamically according to deadline. The earlier the deadline, the higher the priority; the later the deadline, the lower the priority. Under the EDF policy, when a process becomes runnable, it must announce its deadline requirements to the system. Priorities may have to be adjusted to reflect the deadline of the newly runnable process. Note how this differs from rate-monotonic scheduling, where priorities are fixed.

To illustrate EDF scheduling, we again schedule the processes shown in Figure 5.23, which failed to meet deadline requirements under rate-monotonic scheduling. Recall that $P_1$ has values of $p_1 = 50$ and $t_1 = 25$ and that $P_2$ has values of $p_2 = 80$ and $t_2 = 35$. The EDF scheduling of these processes is shown in Figure 5.24. Process $P_1$ has the earliest deadline, so its initial priority is higher than that of process $P_2$. Process $P_2$ begins running at the end of the CPU burst for $P_1$. However, whereas rate-monotonic scheduling allows $P_1$ to preempt $P_2$

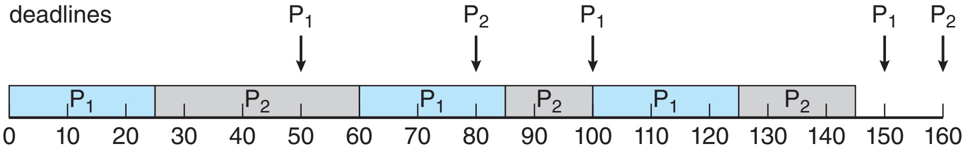


**Figure 5.24** Earliest-deadline-first scheduling.

at the beginning of its next period at time 50, EDF scheduling allows process $P_2$ to continue running. $P_2$ now has a higher priority than $P_1$ because its next deadline (at time 80) is earlier than that of $P_1$ (at time 100). Thus, both $P_1$ and $P_2$ meet their first deadlines. Process $P_1$ again begins running at time 60 and completes its second CPU burst at time 85, also meeting its second deadline at time 100. $P_2$ begins running at this point, only to be preempted by $P_1$ at the start of its next period at time 100. $P_2$ is preempted because $P_1$ has an earlier deadline (time 150) than $P_2$ (time 160). At time 125, $P_1$ completes its CPU burst and $P_2$ resumes execution, finishing at time 145 and meeting its deadline as well. The system is idle until time 150, when $P_1$ is scheduled to run once again.

Unlike the rate-monotonic algorithm, EDF scheduling does not require that processes be periodic, nor must a process require a constant amount of CPU time per burst. The only requirement is that a process announce its deadline to the scheduler when it becomes runnable. The appeal of EDF scheduling is that it is theoretically optimal—theoretically, it can schedule processes so that each process can meet its deadline requirements and CPU utilization will be 100 percent. In practice, however, it is impossible to achieve this level of CPU utilization due to the cost of context switching between processes and interrupt handling.

### 5.6.5 Proportional Share Scheduling

**Proportional share** schedulers operate by allocating $T$ shares among all applications. An application can receive $N$ shares of time, thus ensuring that the application will have $N/T$ of the total processor time. As an example, assume that a total of $T = 100$ shares is to be divided among three processes, $A$, $B$, and $C$. $A$ is assigned 50 shares, $B$ is assigned 15 shares, and $C$ is assigned 20 shares. This scheme ensures that $A$ will have 50 percent of total processor time, $B$ will have 15 percent, and $C$ will have 20 percent.

Proportional share schedulers must work in conjunction with an admission-control policy to guarantee that an application receives its allocated shares of time. An admission-control policy will admit a client requesting a particular number of shares only if sufficient shares are available. In our current example, we have allocated $50 + 15 + 20 = 85$ shares of the total of 100 shares. If a new process $D$ requested 30 shares, the admission controller would deny $D$ entry into the system.

### 5.6.6 POSIX Real-Time Scheduling

The POSIX standard also provides extensions for real-time computing—POSIX.1b. Here, we cover some of the POSIX API related to scheduling real-time threads. POSIX defines two scheduling classes for real-time threads:

- SCHED_FIFO
- SCHED_RR

SCHED_FIFO schedules threads according to a first-come, first-served policy using a FIFO queue as outlined in Section 5.3.1. However, there is no time slicing among threads of equal priority. Therefore, the highest-priority real-time thread at the front of the FIFO queue will be granted the CPU until it terminates or blocks. SCHED_RR uses a round-robin policy. It is similar to SCHED_FIFO except that it provides time slicing among threads of equal priority. POSIX provides an additional scheduling class—SCHED_OTHER—but its implementation is undefined and system specific; it may behave differently on different systems.

The POSIX API specifies the following two functions for getting and setting the scheduling policy:

- `pthread_attr_getschedpolicy(pthread_attr_t *attr, int *policy)`
- `pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy)`

The first parameter to both functions is a pointer to the set of attributes for the thread. The second parameter is either (1) a pointer to an integer that is set to the current scheduling policy (for `pthread_attr_getsched_policy()`) or (2) an integer value (SCHED_FIFO, SCHED_RR, or SCHED_OTHER) for the `pthread_attr_setsched_policy()` function. Both functions return nonzero values if an error occurs.

In Figure 5.25, we illustrate a POSIX Pthread program using this API. This program first determines the current scheduling policy and then sets the scheduling algorithm to SCHED_FIFO.

## 5.7 Operating-System Examples

We turn next to a description of the scheduling policies of the Linux, Windows, and Solaris operating systems. It is important to note that we use the term ***process scheduling*** in a general sense here. In fact, we are describing the scheduling of ***kernel threads*** with Solaris and Windows systems and of ***tasks*** with the Linux scheduler.

### 5.7.1 Example: Linux Scheduling

Process scheduling in Linux has had an interesting history. Prior to Version 2.5, the Linux kernel ran a variation of the traditional UNIX scheduling algorithm. However, as this algorithm was not designed with SMP systems in mind, it did not adequately support systems with multiple processors. In addition, it resulted in poor performance for systems with a large number of runnable processes. With Version 2.5 of the kernel, the scheduler was overhauled to include a scheduling algorithm—known as $O(1)$—that ran in constant time regardless of the number of tasks in the system. The $O(1)$ scheduler also provided

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char *argv[])
{
  int i, policy;
  pthread_t tid[NUM_THREADS];
  pthread_attr_t attr;

  /* get the default attributes */
  pthread_attr_init(&attr);

  /* get the current scheduling policy */
  if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
    fprintf(stderr, "Unable to get policy.\n");
  else {
    if (policy == SCHED_OTHER)
     printf("SCHED_OTHER\n");
    else if (policy == SCHED_RR)
     printf("SCHED_RR\n");
    else if (policy == SCHED_FIFO)
     printf("SCHED_FIFO\n");
  }

  /* set the scheduling policy - FIFO, RR, or OTHER */
  if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
    fprintf(stderr, "Unable to set policy.\n");

  /* create the threads */
  for (i = 0; i < NUM_THREADS; i++)
     pthread_create(&tid[i],&attr,runner,NULL);

  /* now join on each thread */
  for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);
}

/* Each thread will begin control in this function */
void *runner(void *param)
{
  /* do some work ... */

  pthread_exit(0);
}
```

**Figure 5.25** POSIX real-time scheduling API.

increased support for SMP systems, including processor affinity and load balancing between processors. However, in practice, although the *O*(1) scheduler delivered excellent performance on SMP systems, it led to poor response times for the interactive processes that are common on many desktop computer systems. During development of the 2.6 kernel, the scheduler was again revised; and in release 2.6.23 of the kernel, the ***Completely Fair Scheduler*** (CFS) became the default Linux scheduling algorithm.

Scheduling in the Linux system is based on **scheduling classes**. Each class is assigned a specific priority. By using different scheduling classes, the kernel can accommodate different scheduling algorithms based on the needs of the system and its processes. The scheduling criteria for a Linux server, for example, may be different from those for a mobile device running Linux. To decide which task to run next, the scheduler selects the highest-priority task belonging to the highest-priority scheduling class. Standard Linux kernels implement two scheduling classes: (1) a default scheduling class using the CFS scheduling algorithm and (2) a real-time scheduling class. We discuss each of these classes here. New scheduling classes can, of course, be added.

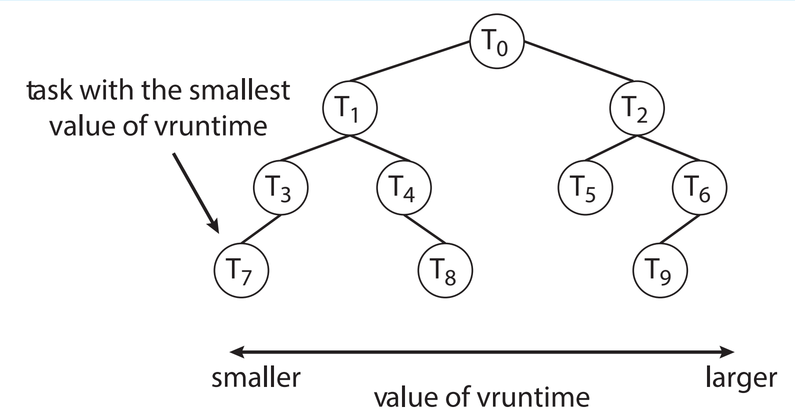
Rather than using strict rules that associate a relative priority value with the length of a time quantum, the CFS scheduler assigns a proportion of CPU processing time to each task. This proportion is calculated based on the **nice value** assigned to each task. Nice values range from −20 to +19, where a numerically lower nice value indicates a higher relative priority. Tasks with lower nice values receive a higher proportion of CPU processing time than tasks with higher nice values. The default nice value is 0. (The term ***nice*** comes from the idea that if a task increases its nice value from, say, 0 to +10, it is being nice to other tasks in the system by lowering its relative priority. In other words, nice processes finish last!) CFS doesn't use discrete values of time slices and instead identifies a **targeted latency**, which is an interval of time during which every runnable task should run at least once. Proportions of CPU time are allocated from the value of targeted latency. In addition to having default and minimum values, targeted latency can increase if the number of active tasks in the system grows beyond a certain threshold.

The CFS scheduler doesn't directly assign priorities. Rather, it records how long each task has run by maintaining the **virtual run time** of each task using the per-task variable `vruntime`. The virtual run time is associated with a decay factor based on the priority of a task: lower-priority tasks have higher rates of decay than higher-priority tasks. For tasks at normal priority (nice values of 0), virtual run time is identical to actual physical run time. Thus, if a task with default priority runs for 200 milliseconds, its `vruntime` will also be 200 milliseconds. However, if a lower-priority task runs for 200 milliseconds, its `vruntime` will be higher than 200 milliseconds. Similarly, if a higher-priority task runs for 200 milliseconds, its `vruntime` will be less than 200 milliseconds. To decide which task to run next, the scheduler simply selects the task that has the smallest `vruntime` value. In addition, a higher-priority task that becomes available to run can preempt a lower-priority task.

Let's examine the CFS scheduler in action: Assume that two tasks have the same nice values. One task is I/O-bound, and the other is CPU-bound. Typically, the I/O-bound task will run only for short periods before blocking for additional I/O, and the CPU-bound task will exhaust its time period whenever it has an opportunity to run on a processor. Therefore, the value of `vruntime` will

### *CFS PERFORMANCE*

The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Rather than using a standard queue data structure, each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below.



When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(log\ N)$ operations (where $N$ is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.

eventually be lower for the I/O-bound task than for the CPU-bound task, giving the I/O-bound task higher priority than the CPU-bound task. At that point, if the CPU-bound task is executing when the I/O-bound task becomes eligible to run (for example, when I/O the task is waiting for becomes available), the I/O-bound task will preempt the CPU-bound task.

Linux also implements real-time scheduling using the POSIX standard as described in Section 5.6.6. Any task scheduled using either the SCHED_FIFO or the SCHED_RR real-time policy runs at a higher priority than normal (non-real-time) tasks. Linux uses two separate priority ranges, one for real-time tasks and a second for normal tasks. Real-time tasks are assigned static priorities within the range of 0 to 99, and normal tasks are assigned priorities from 100 to 139. These two ranges map into a global priority scheme wherein numerically lower values indicate higher relative priorities. Normal tasks are assigned a priority

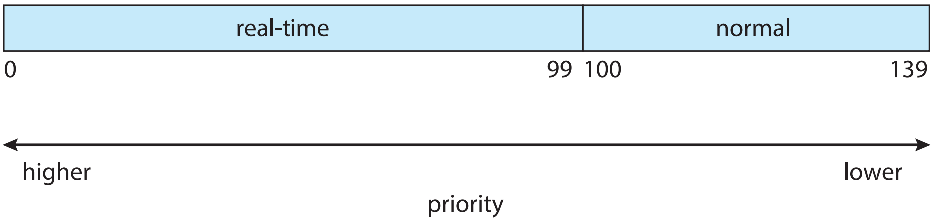


**Figure 5.26** Scheduling priorities on a Linux system.

based on their nice values, where a value of −20 maps to priority 100 and a nice value of +19 maps to 139. This scheme is shown in Figure 5.26.

The CFS scheduler also supports load balancing, using a sophisticated technique that equalizes the load among processing cores yet is also NUMA-aware and minimizes the migration of threads. CFS defines the load of each thread as a combination of the thread's priority and its average rate of CPU utilization. Therefore, a thread that has a high priority, yet is mostly I/O-bound and requires little CPU usage, has a generally low load, similar to the load of a low-priority thread that has high CPU utilization. Using this metric, the load of a queue is the sum of the loads of all threads in the queue, and balancing is simply ensuring that all queues have approximately the same load.

As highlighted in Section 5.5.4, however, migrating a thread may result in a memory access penalty due to either having to invalidate cache contents or, on NUMA systems, incurring longer memory access times. To address this problem, Linux identifies a hierarchical system of scheduling domains. A **scheduling domain** is a set of CPU cores that can be balanced against one another. This idea is illustrated in Figure 5.27. The cores in each scheduling domain are grouped according to how they share the resources of the system. For example, although each core shown in Figure 5.27 may have its own level 1 (L1) cache, pairs of cores share a level 2 (L2) cache and are thus organized into separate $domain_0$ and $domain_1$. Likewise, these two domains may share a level 3 (L3) cache, and are therefore organized into a processor-level domain (also known as a **NUMA node**). Taking this one-step further, on a NUMA system,

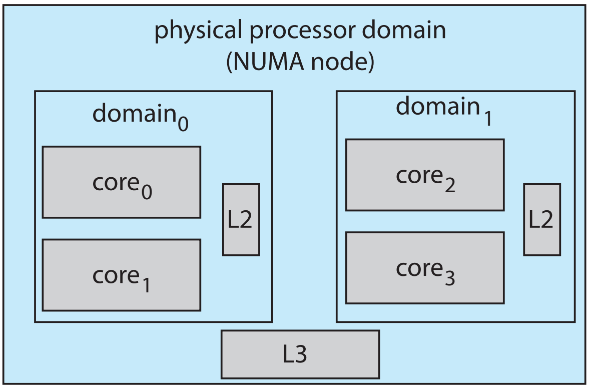


**Figure 5.27** NUMA-aware load balancing with Linux CFS scheduler.

a larger system-level domain would combine separate processor-level NUMA nodes.

The general strategy behind CFS is to balance loads within domains, beginning at the lowest level of the hierarchy. Using Figure 5.27 as an example, initially a thread would only migrate between cores on the same domain (i.e. within $domain_0$ or $domain_1$.) Load balancing at the next level would occur between $domain_0$ and $domain_1$. CFS is reluctant to migrate threads between separate NUMA nodes if a thread would be moved farther from its local memory, and such migration would only occur under severe load imbalances. As a general rule, if the overall system is busy, CFS will not load-balance beyond the domain local to each core to avoid the memory latency penalties of NUMA systems.

### 5.7.2 Example: Windows Scheduling

Windows schedules threads using a priority-based, preemptive scheduling algorithm. The Windows scheduler ensures that the highest-priority thread will always run. The portion of the Windows kernel that handles scheduling is called the **dispatcher**. A thread selected to run by the dispatcher will run until it is preempted by a higher-priority thread, until it terminates, until its time quantum ends, or until it calls a blocking system call, such as for I/O. If a higher-priority real-time thread becomes ready while a lower-priority thread is running, the lower-priority thread will be preempted. This preemption gives a real-time thread preferential access to the CPU when the thread needs such access.

The dispatcher uses a 32-level priority scheme to determine the order of thread execution. Priorities are divided into two classes. The **variable class** contains threads having priorities from 1 to 15, and the **real-time class** contains threads with priorities ranging from 16 to 31. (There is also a thread running at priority 0 that is used for memory management.) The dispatcher uses a queue for each scheduling priority and traverses the set of queues from highest to lowest until it finds a thread that is ready to run. If no ready thread is found, the dispatcher will execute a special thread called the **idle thread**.

There is a relationship between the numeric priorities of the Windows kernel and the Windows API. The Windows API identifies the following six priority classes to which a process can belong:

- IDLE_PRIORITY_CLASS
- BELOW_NORMAL_PRIORITY_CLASS
- NORMAL_PRIORITY_CLASS
- ABOVE_NORMAL_PRIORITY_CLASS
- HIGH_PRIORITY_CLASS
- REALTIME_PRIORITY_CLASS

Processes are typically members of the NORMAL_PRIORITY_CLASS. A process belongs to this class unless the parent of the process was a member of the IDLE_PRIORITY_CLASS or unless another class was specified when the process was created. Additionally, the priority class of a process can be altered with the `SetPriorityClass()` function in the Windows API. Priorities in all classes except the REALTIME_PRIORITY_CLASS are variable, meaning that the priority of a thread belonging to one of these classes can change.

A thread within a given priority class also has a relative priority. The values for relative priorities include:

- IDLE
- LOWEST
- BELOW_NORMAL
- NORMAL
- ABOVE_NORMAL
- HIGHEST
- TIME_CRITICAL

The priority of each thread is based on both the priority class it belongs to and its relative priority within that class. This relationship is shown in Figure 5.28. The values of the priority classes appear in the top row. The left column contains the values for the relative priorities. For example, if the relative priority of a thread in the ABOVE_NORMAL_PRIORITY_CLASS is NORMAL, the numeric priority of that thread is 10.

Furthermore, each thread has a base priority representing a value in the priority range for the class to which the thread belongs. By default, the base priority is the value of the NORMAL relative priority for that class. The base priorities for each priority class are as follows:

- REALTIME_PRIORITY_CLASS—24
- HIGH_PRIORITY_CLASS—13
- ABOVE_NORMAL_PRIORITY_CLASS—10
- NORMAL_PRIORITY_CLASS—8

| | real-time | high | above normal | normal | below normal | idle priority |
|---|---|---|---|---|---|---|
| time-critical | 31 | 15 | 15 | 15 | 15 | 15 |
| highest | 26 | 15 | 12 | 10 | 8 | 6 |
| above normal | 25 | 14 | 11 | 9 | 7 | 5 |
| normal | 24 | 13 | 10 | 8 | 6 | 4 |
| below normal | 23 | 12 | 9 | 7 | 5 | 3 |
| lowest | 22 | 11 | 8 | 6 | 4 | 2 |
| idle | 16 | 1 | 1 | 1 | 1 | 1 |

**Figure 5.28** Windows thread priorities.

- BELOW_NORMAL_PRIORITY_CLASS—6
- IDLE_PRIORITY_CLASS—4

The initial priority of a thread is typically the base priority of the process the thread belongs to, although the `SetThreadPriority()` function in the Windows API can also be used to modify a thread's base priority.

When a thread's time quantum runs out, that thread is interrupted. If the thread is in the variable-priority class, its priority is lowered. The priority is never lowered below the base priority, however. Lowering the priority tends to limit the CPU consumption of compute-bound threads. When a variable-priority thread is released from a wait operation, the dispatcher boosts the priority. The amount of the boost depends on what the thread was waiting for. For example, a thread waiting for keyboard I/O would get a large increase, whereas a thread waiting for a disk operation would get a moderate one. This strategy tends to give good response times to interactive threads that are using the mouse and windows. It also enables I/O-bound threads to keep the I/O devices busy while permitting compute-bound threads to use spare CPU cycles in the background. This strategy is used by several operating systems, including UNIX. In addition, the window with which the user is currently interacting receives a priority boost to enhance its response time.

When a user is running an interactive program, the system needs to provide especially good performance. For this reason, Windows has a special scheduling rule for processes in the NORMAL_PRIORITY_CLASS. Windows distinguishes between the **foreground process** that is currently selected on the screen and the **background processes** that are not currently selected. When a process moves into the foreground, Windows increases the scheduling quantum by some factor—typically by 3. This increase gives the foreground process three times longer to run before a time-sharing preemption occurs.

Windows 7 introduced **user-mode scheduling** (**UMS**), which allows applications to create and manage threads independently of the kernel. Thus, an application can create and schedule multiple threads without involving the Windows kernel scheduler. For applications that create a large number of threads, scheduling threads in user mode is much more efficient than kernel-mode thread scheduling, as no kernel intervention is necessary.

Earlier versions of Windows provided a similar feature known as **fiber** , which allowed several user-mode threads (fibers) to be mapped to a single kernel thread. However, fibers were of limited practical use. A fiber was unable to make calls to the Windows API because all fibers had to share the thread environment block (TEB) of the thread on which they were running. This presented a problem if a Windows API function placed state information into the TEB for one fiber, only to have the information overwritten by a different fiber. UMS overcomes this obstacle by providing each user-mode thread with its own thread context.

In addition, unlike fibers, UMS is not intended to be used directly by the programmer. The details of writing user-mode schedulers can be very challenging, and UMS does not include such a scheduler. Rather, the schedulers come from programming language libraries that build on top of UMS. For example, Microsoft provides **Concurrency Runtime** (ConcRT), a concurrent programming framework for C++ that is designed for task-based parallelism

(Section 4.2) on multicore processors. ConcRT provides a user-mode scheduler together with facilities for decomposing programs into tasks, which can then be scheduled on the available processing cores.

Windows also supports scheduling on multiprocessor systems as described in Section 5.5 by attempting to schedule a thread on the most optimal processing core for that thread, which includes maintaining a thread's preferred as well as most recent processor. One technique used by Windows is to create sets of logical processors (known as **SMT sets**). On a hyper-threaded SMT system, hardware threads belonging to the same CPU core would also belong to the same SMT set. Logical processors are numbered, beginning from 0. As an example, a dual-threaded/quad-core system would contain eight logical processors, consisting of the four SMT sets: {0, 1}, {2, 3}, {4, 5}, and {6, 7}. To avoid cache memory access penalites highlighted in Section 5.5.4, the scheduler attempts to maintain a thread running on logical processors within the same SMT set.

To distribute loads across different logical processors, each thread is assigned an **ideal processor**, which is a number representing a thread's preferred processor. Each process has an initial seed value identifying the ideal CPU for a thread belonging to that process. This seed is incremented for each new thread created by that process, thereby distributing the load across different logical processors. On SMT systems, the increment for the next ideal processor is in the next SMT set. For example, on a dual-threaded/quad-core system, the ideal processors for threads in a specific process would be assigned 0, 2, 4, 6, 0, 2, .... To avoid the situation wherby the first thread for each process is assigned processor 0, processes are assigned different seed values, thereby distributing the load of threads across all physical processing cores in the system. Continuing our example from above, if the seed for a second process were 1, the ideal processors would be assigned in the order 1, 3, 5, 7, 1, 3, and so forth.

### 5.7.3 Example: Solaris Scheduling

Solaris uses priority-based thread scheduling. Each thread belongs to one of six classes:

1. Time sharing (TS)
2. Interactive (IA)
3. Real time (RT)
4. System (SYS)
5. Fair share (FSS)
6. Fixed priority (FP)

Within each class there are different priorities and different scheduling algorithms.

The default scheduling class for a process is time sharing. The scheduling policy for the time-sharing class dynamically alters priorities and assigns time slices of different lengths using a multilevel feedback queue. By default, there is an inverse relationship between priorities and time slices. The higher the

| priority | time quantum | time quantum expired | return from sleep |
|---|---|---|---|
| 0 | 200 | 0 | 50 |
| 5 | 200 | 0 | 50 |
| 10 | 160 | 0 | 51 |
| 15 | 160 | 5 | 51 |
| 20 | 120 | 10 | 52 |
| 25 | 120 | 15 | 52 |
| 30 | 80 | 20 | 53 |
| 35 | 80 | 25 | 54 |
| 40 | 40 | 30 | 55 |
| 45 | 40 | 35 | 56 |
| 50 | 40 | 40 | 58 |
| 55 | 40 | 45 | 58 |
| 59 | 20 | 49 | 59 |

**Figure 5.29** Solaris dispatch table for time-sharing and interactive threads.

priority, the smaller the time slice; and the lower the priority, the larger the time slice. Interactive processes typically have a higher priority; CPU-bound processes, a lower priority. This scheduling policy gives good response time for interactive processes and good throughput for CPU-bound processes. The interactive class uses the same scheduling policy as the time-sharing class, but it gives windowing applications—such as those created by the KDE or GNOME window managers—a higher priority for better performance.

Figure 5.29 shows the simplified dispatch table for scheduling time-sharing and interactive threads. These two scheduling classes include 60 priority levels, but for brevity, we display only a handful. (To see the full dispatch table on a Solaris system or VM, run `dispadmin -c TS -g`.) The dispatch table shown in Figure 5.29 contains the following fields:

- **Priority**. The class-dependent priority for the time-sharing and interactive classes. A higher number indicates a higher priority.
- **Time quantum**. The time quantum for the associated priority. This illustrates the inverse relationship between priorities and time quanta: the lowest priority (priority 0) has the highest time quantum (200 milliseconds), and the highest priority (priority 59) has the lowest time quantum (20 milliseconds).
- **Time quantum expired**. The new priority of a thread that has used its entire time quantum without blocking. Such threads are considered CPU-intensive. As shown in the table, these threads have their priorities lowered.

- **Return from sleep**. The priority of a thread that is returning from sleeping (such as from waiting for I/O). As the table illustrates, when I/O is available for a waiting thread, its priority is boosted to between 50 and 59, supporting the scheduling policy of providing good response time for interactive processes.

Threads in the real-time class are given the highest priority. A real-time process will run before a process in any other class. This assignment allows a real-time process to have a guaranteed response from the system within a bounded period of time. In general, however, few processes belong to the real-time class.

Solaris uses the system class to run kernel threads, such as the scheduler and paging daemon. Once the priority of a system thread is established, it does not change. The system class is reserved for kernel use (user processes running in kernel mode are not in the system class).

The fixed-priority and fair-share classes were introduced with Solaris 9. Threads in the fixed-priority class have the same priority range as those in the time-sharing class; however, their priorities are not dynamically adjusted. The fair-share class uses CPU **shares** instead of priorities to make scheduling decisions. CPU shares indicate entitlement to available CPU resources and are allocated to a set of processes (known as a **project**).

Each scheduling class includes a set of priorities. However, the scheduler converts the class-specific priorities into global priorities and selects the thread with the highest global priority to run. The selected thread runs on the CPU until it (1) blocks, (2) uses its time slice, or (3) is preempted by a higher-priority thread. If there are multiple threads with the same priority, the scheduler uses a round-robin queue. Figure 5.30 illustrates how the six scheduling classes relate to one another and how they map to global priorities. Notice that the kernel maintains ten threads for servicing interrupts. These threads do not belong to any scheduling class and execute at the highest priority (160–169). As mentioned, Solaris has traditionally used the many-to-many model (Section 4.3.3) but switched to the one-to-one model (Section 4.3.2) beginning with Solaris 9.

## 5.8 Algorithm Evaluation

How do we select a CPU-scheduling algorithm for a particular system? As we saw in Section 5.3, there are many scheduling algorithms, each with its own parameters. As a result, selecting an algorithm can be difficult.

The first problem is defining the criteria to be used in selecting an algorithm. As we saw in Section 5.2, criteria are often defined in terms of CPU utilization, response time, or throughput. To select an algorithm, we must first define the relative importance of these elements. Our criteria may include several measures, such as these:

- Maximizing CPU utilization under the constraint that the maximum response time is 300 milliseconds

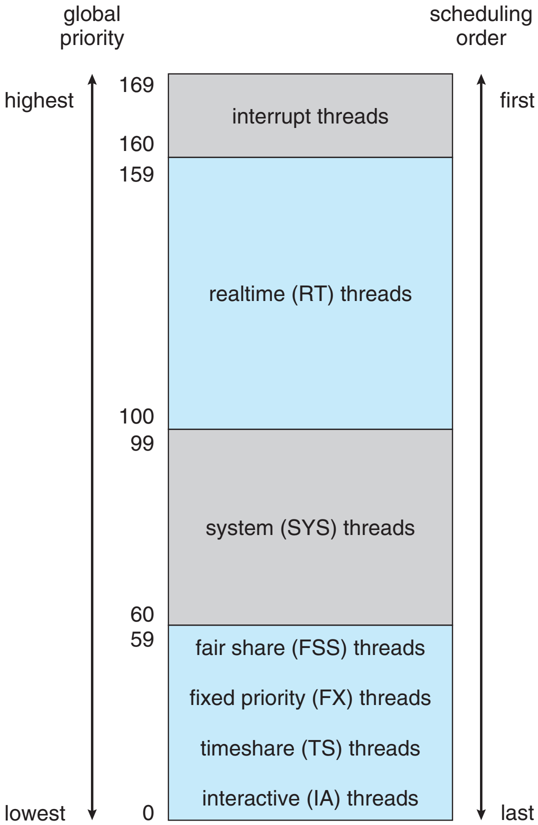

global priority
scheduling order
highest
lowest
first
last
169
160
159
100
99
60
59
0
interrupt threads
realtime (RT) threads
system (SYS) threads
fair share (FSS) threads
fixed priority (FX) threads
timeshare (TS) threads
interactive (IA) threads

**Figure 5.30** Solaris scheduling.

- Maximizing throughput such that turnaround time is (on average) linearly proportional to total execution time

Once the selection criteria have been defined, we want to evaluate the algorithms under consideration. We next describe the various evaluation methods we can use.

### 5.8.1 Deterministic Modeling

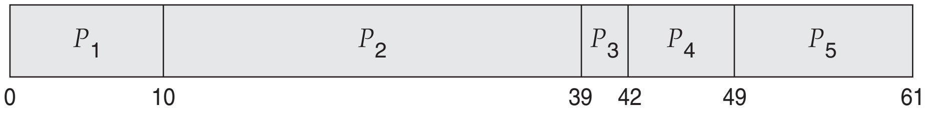
One major class of evaluation methods is **analytic evaluation**. Analytic evaluation uses the given algorithm and the system workload to produce a formula or number to evaluate the performance of the algorithm for that workload.

**Deterministic modeling** is one type of analytic evaluation. This method takes a particular predetermined workload and defines the performance of each algorithm for that workload. For example, assume that we have the workload shown below. All five processes arrive at time 0, in the order given, with the length of the CPU burst given in milliseconds:

| Process | Burst Time |
|---|---|
| $P_1$ | 10 |
| $P_2$ | 29 |
| $P_3$ | 3 |
| $P_4$ | 7 |
| $P_5$ | 12 |

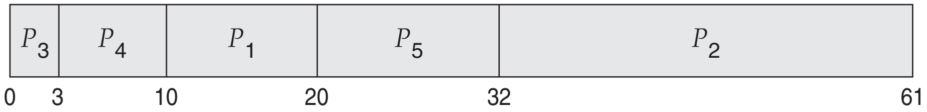
Consider the FCFS, SJF, and RR (quantum = 10 milliseconds) scheduling algorithms for this set of processes. Which algorithm would give the minimum average waiting time?

For the FCFS algorithm, we would execute the processes as



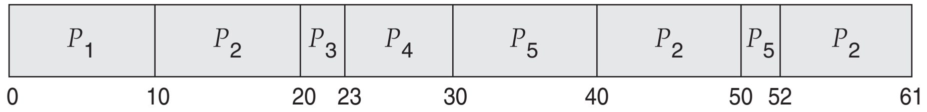
The waiting time is 0 milliseconds for process $P_1$, 10 milliseconds for process $P_2$, 39 milliseconds for process $P_3$, 42 milliseconds for process $P_4$, and 49 milliseconds for process $P_5$. Thus, the average waiting time is (0 + 10 + 39 + 42 + 49)/5 = 28 milliseconds.

With nonpreemptive SJF scheduling, we execute the processes as



The waiting time is 10 milliseconds for process $P_1$, 32 milliseconds for process $P_2$, 0 milliseconds for process $P_3$, 3 milliseconds for process $P_4$, and 20 milliseconds for process $P_5$. Thus, the average waiting time is (10 + 32 + 0 + 3 + 20)/5 = 13 milliseconds.

With the RR algorithm, we execute the processes as



The waiting time is 0 milliseconds for process $P_1$, 32 milliseconds for process $P_2$, 20 milliseconds for process $P_3$, 23 milliseconds for process $P_4$, and 40 milliseconds for process $P_5$. Thus, the average waiting time is (0 + 32 + 20 + 23 + 40)/5 = 23 milliseconds.

We can see that, in this case, the average waiting time obtained with the SJF policy is less than half that obtained with FCFS scheduling; the RR algorithm gives us an intermediate value.

Deterministic modeling is simple and fast. It gives us exact numbers, allowing us to compare the algorithms. However, it requires exact numbers for input, and its answers apply only to those cases. The main uses of deterministic modeling are in describing scheduling algorithms and providing examples. In cases where we are running the same program over and over again and can measure the program's processing requirements exactly, we may be able to use deterministic modeling to select a scheduling algorithm. Furthermore, over a set of examples, deterministic modeling may indicate trends that can then be analyzed and proved separately. For example, it can be shown that, for the environment described (all processes and their times available at time 0), the SJF policy will always result in the minimum waiting time.

### 5.8.2 Queueing Models

On many systems, the processes that are run vary from day to day, so there is no static set of processes (or times) to use for deterministic modeling. What can be determined, however, is the distribution of CPU and I/O bursts. These distributions can be measured and then approximated or simply estimated. The result is a mathematical formula describing the probability of a particular CPU burst. Commonly, this distribution is exponential and is described by its mean. Similarly, we can describe the distribution of times when processes arrive in the system (the arrival-time distribution). From these two distributions, it is possible to compute the average throughput, utilization, waiting time, and so on for most algorithms.

The computer system is described as a network of servers. Each server has a queue of waiting processes. The CPU is a server with its ready queue, as is the I/O system with its device queues. Knowing arrival rates and service rates, we can compute utilization, average queue length, average wait time, and so on. This area of study is called **queueing-network analysis**.

As an example, let $n$ be the average long-term queue length (excluding the process being serviced), let $W$ be the average waiting time in the queue, and let $\lambda$ be the average arrival rate for new processes in the queue (such as three processes per second). We expect that during the time $W$ that a process waits, $\lambda \times W$ new processes will arrive in the queue. If the system is in a steady state, then the number of processes leaving the queue must be equal to the number of processes that arrive. Thus,

$$n = \lambda \times W.$$

This equation, known as **Little's formula**, is particularly useful because it is valid for any scheduling algorithm and arrival distribution. For example $n$ could be the number of customers in a store.

We can use Little's formula to compute one of the three variables if we know the other two. For example, if we know that 7 processes arrive every second (on average) and that there are normally 14 processes in the queue, then we can compute the average waiting time per process as 2 seconds.

Queueing analysis can be useful in comparing scheduling algorithms, but it also has limitations. At the moment, the classes of algorithms and distributions that can be handled are fairly limited. The mathematics of complicated algorithms and distributions can be difficult to work with. Thus, arrival and service distributions are often defined in mathematically tractable—but unrealistic—ways. It is also generally necessary to make a number of independent assumptions, which may not be accurate. As a result of these difficulties, queueing models are often only approximations of real systems, and the accuracy of the computed results may be questionable.

### 5.8.3 Simulations

To get a more accurate evaluation of scheduling algorithms, we can use simulations. Running simulations involves programming a model of the computer system. Software data structures represent the major components of the system. The simulator has a variable representing a clock. As this variable's value is increased, the simulator modifies the system state to reflect the activities of the devices, the processes, and the scheduler. As the simulation executes, statistics that indicate algorithm performance are gathered and printed.

The data to drive the simulation can be generated in several ways. The most common method uses a random-number generator that is programmed to generate processes, CPU burst times, arrivals, departures, and so on, according to probability distributions. The distributions can be defined mathematically (uniform, exponential, Poisson) or empirically. If a distribution is to be defined empirically, measurements of the actual system under study are taken. The results define the distribution of events in the real system; this distribution can then be used to drive the simulation.

A distribution-driven simulation may be inaccurate, however, because of relationships between successive events in the real system. The frequency distribution indicates only how many instances of each event occur; it does not indicate anything about the order of their occurrence. To correct this problem, we can use **trace files**. We create a trace by monitoring the real system and recording the sequence of actual events (Figure 5.31). We then use this sequence to drive the simulation. Trace files provide an excellent way to compare two algorithms on exactly the same set of real inputs. This method can produce accurate results for its inputs.

Simulations can be expensive, often requiring many hours of computer time. A more detailed simulation provides more accurate results, but it also

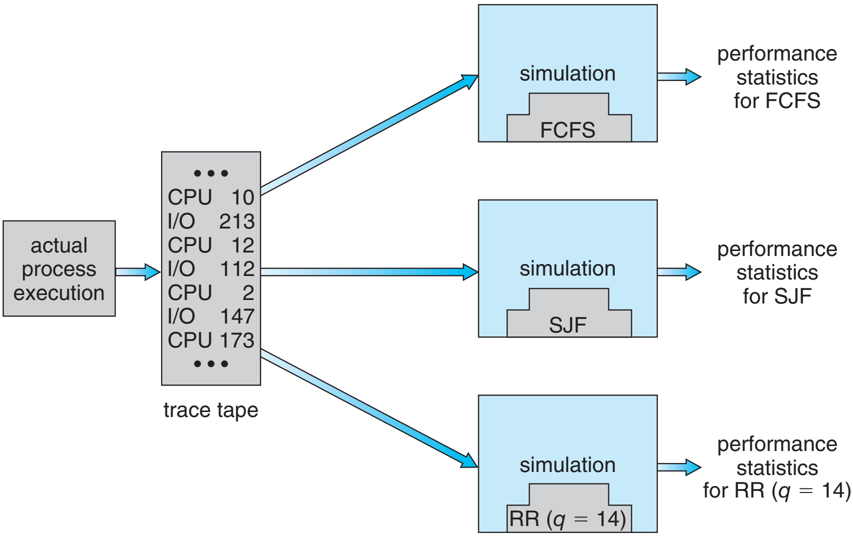

actual process execution
• • •
CPU 10
I/O 213
CPU 12
I/O 112
CPU 2
I/O 147
CPU 173
• • •
trace tape
simulation
FCFS
performance statistics for FCFS
simulation
SJF
performance statistics for SJF
simulation
RR (q = 14)
performance statistics for RR (q = 14)

**Figure 5.31** Evaluation of CPU schedulers by simulation.

takes more computer time. In addition, trace files can require large amounts of storage space. Finally, the design, coding, and debugging of the simulator can be a major task.

### 5.8.4 Implementation

Even a simulation is of limited accuracy. The only completely accurate way to evaluate a scheduling algorithm is to code it up, put it in the operating system, and see how it works. This approach puts the actual algorithm in the real system for evaluation under real operating conditions.

This method is not without expense. The expense is incurred in coding the algorithm and modifying the operating system to support it (along with its required data structures). There is also cost in testing the changes, usually in virtual machines rather than on dedicated hardware. **Regression testing** confirms that the changes haven't made anything worse, and haven't caused new bugs or caused old bugs to be recreated (for example because the algorithm being replaced solved some bug and changing it caused that bug to reoccur).

Another difficulty is that the environment in which the algorithm is used will change. The environment will change not only in the usual way, as new programs are written and the types of problems change, but also as a result of the performance of the scheduler. If short processes are given priority, then users may break larger processes into sets of smaller processes. If interactive processes are given priority over noninteractive processes, then users may switch to interactive use. This problem is usually addressed by using tools or scripts that encapsulate complete sets of actions, repeatedly using those tools, and using those tools while measuring the results (and detecting any problems they cause in the new environment).

Of course human or program behavior can attempt to circumvent scheduling algorithms. For example, researchers designed one system that classified interactive and noninteractive processes automatically by looking at the amount of terminal I/O. If a process did not input or output to the terminal in a 1-second interval, the process was classified as noninteractive and was moved to a lower-priority queue. In response to this policy, one programmer modified his programs to write an arbitrary character to the terminal at regular intervals of less than 1 second. The system gave his programs a high priority, even though the terminal output was completely meaningless.

In general, most flexible scheduling algorithms are those that can be altered by the system managers or by the users so that they can be tuned for a specific application or set of applications. A workstation that performs high-end graphical applications, for instance, may have scheduling needs different from those of a web server or file server. Some operating systems—particularly several versions of UNIX—allow the system manager to fine-tune the scheduling parameters for a particular system configuration. For example, Solaris provides the `dispadmin` command to allow the system administrator to modify the parameters of the scheduling classes described in Section 5.7.3.

Another approach is to use APIs that can modify the priority of a process or thread. The Java, POSIX, and Windows APIs provide such functions. The downfall of this approach is that performance-tuning a system or application most often does not result in improved performance in more general situations.

## 5.9 Summary

- CPU scheduling is the task of selecting a waiting process from the ready queue and allocating the CPU to it. The CPU is allocated to the selected process by the dispatcher.
- Scheduling algorithms may be either preemptive (where the CPU can be taken away from a process) or nonpreemptive (where a process must voluntarily relinquish control of the CPU). Almost all modern operating systems are preemptive.
- Scheduling algorithms can be evaluated according to the following five criteria: (1) CPU utilization, (2) throughput, (3) turnaround time, (4) waiting time, and (5) response time.
- First-come, first-served (FCFS) scheduling is the simplest scheduling algorithm, but it can cause short processes to wait for very long processes.
- Shortest-job-first (SJF) scheduling is provably optimal, providing the shortest average waiting time. Implementing SJF scheduling is difficult, however, because predicting the length of the next CPU burst is difficult.
- Round-robin (RR) scheduling allocates the CPU to each process for a time quantum. If the process does not relinquish the CPU before its time quantum expires, the process is preempted, and another process is scheduled to run for a time quantum.
- Priority scheduling assigns each process a priority, and the CPU is allocated to the process with the highest priority. Processes with the same priority can be scheduled in FCFS order or using RR scheduling.
- Multilevel queue scheduling partitions processes into several separate queues arranged by priority, and the scheduler executes the processes in the highest-priority queue. Different scheduling algorithms may be used in each queue.
- Multilevel feedback queues are similar to multilevel queues, except that a process may migrate between different queues.
- Multicore processors place one or more CPUs on the same physical chip, and each CPU may have more than one hardware thread. From the perspective of the operating system, each hardware thread appears to be a logical CPU.
- Load balancing on multicore systems equalizes loads between CPU cores, although migrating threads between cores to balance loads may invalidate cache contents and therefore may increase memory access times.
- Soft real-time scheduling gives priority to real-time tasks over non-real-time tasks. Hard real-time scheduling provides timing guarantees for real-time tasks,
- Rate-monotonic real-time scheduling schedules periodic tasks using a static priority policy with preemption.

- Earliest-deadline-first (EDF) scheduling assigns priorities according to deadline. The earlier the deadline, the higher the priority; the later the deadline, the lower the priority.
- Proportional share scheduling allocates $T$ shares among all applications. If an application is allocated $N$ shares of time, it is ensured of having $N/T$ of the total processor time.
- Linux uses the completely fair scheduler (CFS), which assigns a proportion of CPU processing time to each task. The proportion is based on the virtual runtime (`vruntime`) value associated with each task.
- Windows scheduling uses a preemptive, 32-level priority scheme to determine the order of thread scheduling.
- Solaris identifies six unique scheduling classes that are mapped to a global priority. CPU-intensive threads are generally assigned lower priorities (and longer time quantums), and I/O-bound threads are usually assigned higher priorities (with shorter time quantums.)
- Modeling and simulations can be used to evaluate a CPU scheduling algorithm.

## Practice Exercises

**5.1** A CPU-scheduling algorithm determines an order for the execution of its scheduled processes. Given $n$ processes to be scheduled on one processor, how many different schedules are possible? Give a formula in terms of $n$.

**5.2** Explain the difference between preemptive and nonpreemptive scheduling.

**5.3** Suppose that the following processes arrive for execution at the times indicated. Each process will run for the amount of time listed. In answering the questions, use nonpreemptive scheduling, and base all decisions on the information you have at the time the decision must be made.

| Process | Arrival Time | Burst Time |
|---|---|---|
| $P_1$ | 0.0 | 8 |
| $P_2$ | 0.4 | 4 |
| $P_3$ | 1.0 | 1 |

a. What is the average turnaround time for these processes with the FCFS scheduling algorithm?

b. What is the average turnaround time for these processes with the SJF scheduling algorithm?

c. The SJF algorithm is supposed to improve performance, but notice that we chose to run process $P_1$ at time 0 because we did not know that two shorter processes would arrive soon. Compute what the average turnaround time will be if the CPU is left idle for the first 1 unit and then SJF scheduling is used. Remember that processes $P_1$ and $P_2$ are waiting during this idle time, so their waiting time may increase. This algorithm could be known as ***future-knowledge scheduling***.

**5.4** Consider the following set of processes, with the length of the CPU burst time given in milliseconds:

| Process | Burst Time | Priority |
|---|---|---|
| $P_1$ | 2 | 2 |
| $P_2$ | 1 | 1 |
| $P_3$ | 8 | 4 |
| $P_4$ | 4 | 2 |
| $P_5$ | 5 | 3 |

The processes are assumed to have arrived in the order $P_1$, $P_2$, $P_3$, $P_4$, $P_5$, all at time 0.

a. Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, nonpreemptive priority (a larger priority number implies a higher priority), and RR (quantum = 2).

b. What is the turnaround time of each process for each of the scheduling algorithms in part a?

c. What is the waiting time of each process for each of these scheduling algorithms?

d. Which of the algorithms results in the minimum average waiting time (over all processes)?

**5.5** The following processes are being scheduled using a preemptive, round-robin scheduling algorithm.

| Process | Priority | Burst | Arrival |
|---|---|---|---|
| $P_1$ | 40 | 20 | 0 |
| $P_2$ | 30 | 25 | 25 |
| $P_3$ | 30 | 25 | 30 |
| $P_4$ | 35 | 15 | 60 |
| $P_5$ | 5 | 10 | 100 |
| $P_6$ | 10 | 10 | 105 |

Each process is assigned a numerical priority, with a higher number indicating a higher relative priority. In addition to the processes listed below, the system also has an **idle task** (which consumes no CPU resources and is identified as $P_{idle}$). This task has priority 0 and is scheduled whenever the system has no other available processes to run. The length of a time quantum is 10 units. If a process is preempted by a higher-priority process, the preempted process is placed at the end of the queue.

a. Show the scheduling order of the processes using a Gantt chart.

b. What is the turnaround time for each process?

c. What is the waiting time for each process?

d. What is the CPU utilization rate?

**5.6** What advantage is there in having different time-quantum sizes at different levels of a multilevel queueing system?

**5.7** Many CPU-scheduling algorithms are parameterized. For example, the RR algorithm requires a parameter to indicate the time slice. Multilevel feedback queues require parameters to define the number of queues, the scheduling algorithms for each queue, the criteria used to move processes between queues, and so on.

These algorithms are thus really sets of algorithms (for example, the set of RR algorithms for all time slices, and so on). One set of algorithms may include another (for example, the FCFS algorithm is the RR algorithm with an infinite time quantum). What (if any) relation holds between the following pairs of algorithm sets?

a. Priority and SJF

b. Multilevel feedback queues and FCFS

c. Priority and FCFS

d. RR and SJF

**5.8** Suppose that a CPU scheduling algorithm favors those processes that have used the least processor time in the recent past. Why will this algorithm favor I/O-bound programs and yet not permanently starve CPU-bound programs?

**5.9** Distinguish between PCS and SCS scheduling.

**5.10** The traditional UNIX scheduler enforces an inverse relationship between priority numbers and priorities: the higher the number, the lower the priority. The scheduler recalculates process priorities once per second using the following function:

Priority = (recent CPU usage / 2) + base

where base = 60 and *recent CPU usage* refers to a value indicating how often a process has used the CPU since priorities were last recalculated.

Assume that recent CPU usage for process $P_1$ is 40, for process $P_2$ is 18, and for process $P_3$ is 10. What will be the new priorities for these three processes when priorities are recalculated? Based on this information, does the traditional UNIX scheduler raise or lower the relative priority of a CPU-bound process?

## Further Reading

Scheduling policies used in the UNIX FreeBSD 5.2 are presented by [McKusick et al. (2015)]; The Linux CFS scheduler is further described in https://www.ibm.com/developerworks/library/l-completely-fair-scheduler/. Solaris scheduling is described by [Mauro and McDougall (2007)]. [Russinovich et al. (2017)] discusses scheduling in Windows internals. [Butenhof (1997)] and [Lewis and Berg (1998)] describe scheduling in Pthreads systems. Multicore scheduling is examined in [McNairy and Bhatia (2005)], [Kongetira et al. (2005)], and [Siddha et al. (2007)] .

## Bibliography

**[Butenhof (1997)]** D. Butenhof, *Programming with POSIX Threads*, Addison-Wesley (1997).

**[Kongetira et al. (2005)]** P. Kongetira, K. Aingaran, and K. Olukotun, “Niagara: A 32-Way Multithreaded SPARC Processor”, *IEEE Micro Magazine*, Volume 25, Number 2 (2005), pages 21–29.

**[Lewis and Berg (1998)]** B. Lewis and D. Berg, *Multithreaded Programming with Pthreads*, Sun Microsystems Press (1998).

**[Mauro and McDougall (2007)]** J. Mauro and R. McDougall, *Solaris Internals: Core Kernel Architecture*, Prentice Hall (2007).

**[McKusick et al. (2015)]** M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson, *The Design and Implementation of the FreeBSD UNIX Operating System–Second Edition*, Pearson (2015).

**[McNairy and Bhatia (2005)]** C. McNairy and R. Bhatia, “Montecito: A Dual–Core, Dual-Threaded Itanium Processor”, *IEEE Micro Magazine*, Volume 25, Number 2 (2005), pages 10–20.

**[Russinovich et al. (2017)]** M. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals–Part 1*, Seventh Edition, Microsoft Press (2017).

**[Siddha et al. (2007)]** S. Siddha, V. Pallipadi, and A. Mallick, “Process Scheduling Challenges in the Era of Multi-Core Processors”, *Intel Technology Journal*, Volume 11, Number 4 (2007).

## Chapter 5 Exercises

**5.11** Of these two types of programs:

a. I/O-bound

b. CPU-bound

which is more likely to have voluntary context switches, and which is more likely to have nonvoluntary context switches? Explain your answer.

**5.12** Discuss how the following pairs of scheduling criteria conflict in certain settings.

a. CPU utilization and response time

b. Average turnaround time and maximum waiting time

c. I/O device utilization and CPU utilization

**5.13** One technique for implementing **lottery scheduling** works by assigning processes lottery tickets, which are used for allocating CPU time. Whenever a scheduling decision has to be made, a lottery ticket is chosen at random, and the process holding that ticket gets the CPU. The BTV operating system implements lottery scheduling by holding a lottery 50 times each second, with each lottery winner getting 20 milliseconds of CPU time (20 milliseconds × 50 = 1 second). Describe how the BTV scheduler can ensure that higher-priority threads receive more attention from the CPU than lower-priority threads.

**5.14** Most scheduling algorithms maintain a **run queue**, which lists processes eligible to run on a processor. On multicore systems, there are two general options: (1) each processing core has its own run queue, or (2) a single run queue is shared by all processing cores. What are the advantages and disadvantages of each of these approaches?

**5.15** Consider the exponential average formula used to predict the length of the next CPU burst. What are the implications of assigning the following values to the parameters used by the algorithm?

a. $\alpha = 0$ and $\tau_0 = 100$ milliseconds

b. $\alpha = 0.99$ and $\tau_0 = 10$ milliseconds

**5.16** A variation of the round-robin scheduler is the **regressive round-robin** scheduler. This scheduler assigns each process a time quantum and a priority. The initial value of a time quantum is 50 milliseconds. However, every time a process has been allocated the CPU and uses its entire time quantum (does not block for I/O), 10 milliseconds is added to its time quantum, and its priority level is boosted. (The time quantum for a process can be increased to a maximum of 100 milliseconds.) When a process blocks before using its entire time quantum, its time quantum is reduced by 5 milliseconds, but its priority remains the same. What type of process (CPU-bound or I/O-bound) does the regressive round-robin scheduler favor? Explain.

**5.17** Consider the following set of processes, with the length of the CPU burst given in milliseconds:

| Process | Burst Time | Priority |
|---|---|---|
| $P_1$ | 5 | 4 |
| $P_2$ | 3 | 1 |
| $P_3$ | 1 | 2 |
| $P_4$ | 7 | 2 |
| $P_5$ | 4 | 3 |

The processes are assumed to have arrived in the order $P_1, P_2, P_3, P_4, P_5$, all at time 0.

a. Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, nonpreemptive priority (a larger priority number implies a higher priority), and RR (quantum = 2).

b. What is the turnaround time of each process for each of the scheduling algorithms in part a?

c. What is the waiting time of each process for each of these scheduling algorithms?

d. Which of the algorithms results in the minimum average waiting time (over all processes)?

**5.18** The following processes are being scheduled using a preemptive, priority-based, round-robin scheduling algorithm.

| Process | Priority | Burst | Arrival |
|---|---|---|---|
| $P_1$ | 8 | 15 | 0 |
| $P_2$ | 3 | 20 | 0 |
| $P_3$ | 4 | 20 | 20 |
| $P_4$ | 4 | 20 | 25 |
| $P_5$ | 5 | 5 | 45 |
| $P_6$ | 5 | 15 | 55 |

Each process is assigned a numerical priority, with a higher number indicating a higher relative priority. The scheduler will execute the highest-priority process. For processes with the same priority, a round-robin scheduler will be used with a time quantum of 10 units. If a process is preempted by a higher-priority process, the preempted process is placed at the end of the queue.

a. Show the scheduling order of the processes using a Gantt chart.

b. What is the turnaround time for each process?

c. What is the waiting time for each process?

**5.19** The `nice` command is used to set the nice value of a process on Linux, as well as on other UNIX systems. Explain why some systems may allow any user to assign a process a nice value >= 0 yet allow only the root (or administrator) user to assign nice values < 0.

**5.20** Which of the following scheduling algorithms could result in starvation?

a. First-come, first-served

b. Shortest job first

c. Round robin

d. Priority

**5.21** Consider a variant of the RR scheduling algorithm in which the entries in the ready queue are pointers to the PCBs.

a. What would be the effect of putting two pointers to the same process in the ready queue?

b. What would be two major advantages and two disadvantages of this scheme?

c. How would you modify the basic RR algorithm to achieve the same effect without the duplicate pointers?

**5.22** Consider a system running ten I/O-bound tasks and one CPU-bound task. Assume that the I/O-bound tasks issue an I/O operation once for every millisecond of CPU computing and that each I/O operation takes 10 milliseconds to complete. Also assume that the context-switching overhead is 0.1 millisecond and that all processes are long-running tasks. Describe the CPU utilization for a round-robin scheduler when:

a. The time quantum is 1 millisecond

b. The time quantum is 10 milliseconds

**5.23** Consider a system implementing multilevel queue scheduling. What strategy can a computer user employ to maximize the amount of CPU time allocated to the user's process?

**5.24** Consider a preemptive priority scheduling algorithm based on dynamically changing priorities. Larger priority numbers imply higher priority. When a process is waiting for the CPU (in the ready queue, but not running), its priority changes at a rate $\alpha$. When it is running, its priority changes at a rate $\beta$. All processes are given a priority of 0 when they enter the ready queue. The parameters $\alpha$ and $\beta$ can be set to give many different scheduling algorithms.

a. What is the algorithm that results from $\beta > \alpha > 0$?

b. What is the algorithm that results from $\alpha < \beta < 0$?

**5.25** Explain the how the following scheduling algorithms discriminate either in favor of or against short processes:

a. FCFS

b. RR

c. Multilevel feedback queues

**5.26** Describe why a shared ready queue might suffer from performance problems in an SMP environment.

**5.27** Consider a load-balancing algorithm that ensures that each queue has approximately the same number of threads, independent of priority. How effectively would a priority-based scheduling algorithm handle this situation if one run queue had all high-priority threads and a second queue had all low-priority threads?

**5.28** Assume that an SMP system has private, per-processor run queues. When a new process is created, it can be placed in either the same queue as the parent process or a separate queue.

a. What are the benefits of placing the new process in the same queue as its parent?

b. What are the benefits of placing the new process in a different queue?

**5.29** Assume that a thread has blocked for network I/O and is eligible to run again. Describe why a NUMA-aware scheduling algorithm should reschedule the thread on the same CPU on which it previously ran.

**5.30** Using the Windows scheduling algorithm, determine the numeric priority of each of the following threads.

a. A thread in the REALTIME_PRIORITY_CLASS with a relative priority of NORMAL

b. A thread in the ABOVE_NORMAL_PRIORITY_CLASS with a relative priority of HIGHEST

c. A thread in the BELOW_NORMAL_PRIORITY_CLASS with a relative priority of ABOVE_NORMAL

**5.31** Assuming that no threads belong to the REALTIME_PRIORITY_CLASS and that none may be assigned a TIME_CRITICAL priority, what combination of priority class and priority corresponds to the highest possible relative priority in Windows scheduling?

**5.32** Consider the scheduling algorithm in the Solaris operating system for time-sharing threads.

a. What is the time quantum (in milliseconds) for a thread with priority 15? With priority 40?

b. Assume that a thread with priority 50 has used its entire time quantum without blocking. What new priority will the scheduler assign this thread?

c. Assume that a thread with priority 20 blocks for I/O before its time quantum has expired. What new priority will the scheduler assign this thread?

**5.33** Assume that two tasks, *A* and *B*, are running on a Linux system. The nice values of *A* and *B* are −5 and +5, respectively. Using the CFS scheduler as a guide, describe how the respective values of `vruntime` vary between the two processes given each of the following scenarios:

- Both *A* and *B* are CPU-bound.
- *A* is I/O-bound, and *B* is CPU-bound.
- *A* is CPU-bound, and *B* is I/O-bound.

**5.34** Provide a specific circumstance that illustrates where rate-monotonic scheduling is inferior to earliest-deadline-first scheduling in meeting real-time process deadlines?

**5.35** Consider two processes, $P_1$ and $P_2$, where $p_1 = 50$, $t_1 = 25$, $p_2 = 75$, and $t_2 = 30$.

a. Can these two processes be scheduled using rate-monotonic scheduling? Illustrate your answer using a Gantt chart such as the ones in Figure 5.21–Figure 5.24.

b. Illustrate the scheduling of these two processes using earliest-deadline-first (EDF) scheduling.

**5.36** Explain why interrupt and dispatch latency times must be bounded in a hard real-time system.

**5.37** Describe the advantages of using heterogeneous multiprocessing in a mobile system.

## Programming Projects

### Scheduling Algorithms

This project involves implementing several different process scheduling algorithms. The scheduler will be assigned a predefined set of tasks and will schedule the tasks based on the selected scheduling algorithm. Each task is assigned a priority and CPU burst. The following scheduling algorithms will be implemented:

- First-come, first-served (FCFS), which schedules tasks in the order in which they request the CPU.
- Shortest-job-first (SJF), which schedules tasks in order of the length of the tasks' next CPU burst.
- Priority scheduling, which schedules tasks based on priority.
- Round-robin (RR) scheduling, where each task is run for a time quantum (or for the remainder of its CPU burst).
- Priority with round-robin, which schedules tasks in order of priority and uses round-robin scheduling for tasks with equal priority.

Priorities range from 1 to 10, where a higher numeric value indicates a higher relative priority. For round-robin scheduling, the length of a time quantum is 10 milliseconds.

#### I. Implementation

The implementation of this project may be completed in either C or Java, and program files supporting both of these languages are provided in the source code download for the text. These supporting files read in the schedule of tasks, insert the tasks into a list, and invoke the scheduler.

The schedule of tasks has the form [***task name***] [***priority***] [***CPU burst***], with the following example format:

```
T1, 4, 20
T2, 2, 25
T3, 3, 25
T4, 3, 15
T5, 10, 10
```

Thus, task `T1` has priority 4 and a CPU burst of 20 milliseconds, and so forth. It is assumed that all tasks arrive at the same time, so your scheduler algorithms do not have to support higher-priority processes preempting processes with lower priorities. In addition, tasks do not have to be placed into a queue or list in any particular order.

There are a few different strategies for organizing the list of tasks, as first presented in Section 5.1.2. One approach is to place all tasks in a single unordered list, where the strategy for task selection depends on the scheduling algorithm. For example, SJF scheduling would search the list to find the task with the shortest next CPU burst. Alternatively, a list could be ordered according to scheduling criteria (that is, by priority). One other strategy involves having a separate queue for each unique priority, as shown in Figure 5.7. These approaches are briefly discussed in Section 5.3.6. It is also worth highlighting that we are using the terms ***list*** and ***queue*** somewhat interchangeably. However, a queue has very specific FIFO functionality, whereas a list does not have such strict insertion and deletion requirements. You are likely to find the functionality of a general list to be more suitable when completing this project.

## II. C Implementation Details

The file `driver.c` reads in the schedule of tasks, inserts each task into a linked list, and invokes the process scheduler by calling the `schedule()` function. The `schedule()` function executes each task according to the specified scheduling algorithm. Tasks selected for execution on the CPU are determined by the `pickNextTask()` function and are executed by invoking the `run()` function defined in the `CPU.c` file. A `Makefile` is used to determine the specific scheduling algorithm that will be invoked by `driver`. For example, to build the FCFS scheduler, we would enter

```
make fcfs
```

and would execute the scheduler (using the schedule of tasks `schedule.txt`) as follows:

```
./fcfs schedule.txt
```

Refer to the `README` file in the source code download for further details. Before proceeding, be sure to familiarize yourself with the source code provided as well as the `Makefile`.

## III. Java Implementation Details

The file `Driver.java` reads in the schedule of tasks, inserts each task into a Java `ArrayList`, and invokes the process scheduler by calling the `schedule()` method. The following interface identifies a generic scheduling algorithm, which the five different scheduling algorithms will implement:

```
public interface Algorithm
{
  // Implementation of scheduling algorithm
  public void schedule();

  // Selects the next task to be scheduled
  public Task pickNetTask();
}
```

The `schedule()` method obtains the next task to be run on the CPU by invoking the `pickNextTask()` method and then executes this `Task` by calling the `static run()` method in the `CPU.java` class.

The program is run as follows:

```
java Driver fcfs schedule.txt
```

Refer to the `README` file in the source code download for further details. Before proceeding, be sure to familiarize yourself with all Java source files provided in the source code download.

## IV. Further Challenges

Two additional challenges are presented for this project:

1. Each task provided to the scheduler is assigned a unique task (`tid`). If a scheduler is running in an SMP environment where each CPU is separately running its own scheduler, there is a possible race condition on the variable that is used to assign task identifiers. Fix this race condition using an atomic integer.

   On Linux and macOS systems, the `__sync_fetch_and_add()` function can be used to atomically increment an integer value. As an example, the following code sample atomically increments `value` by 1:

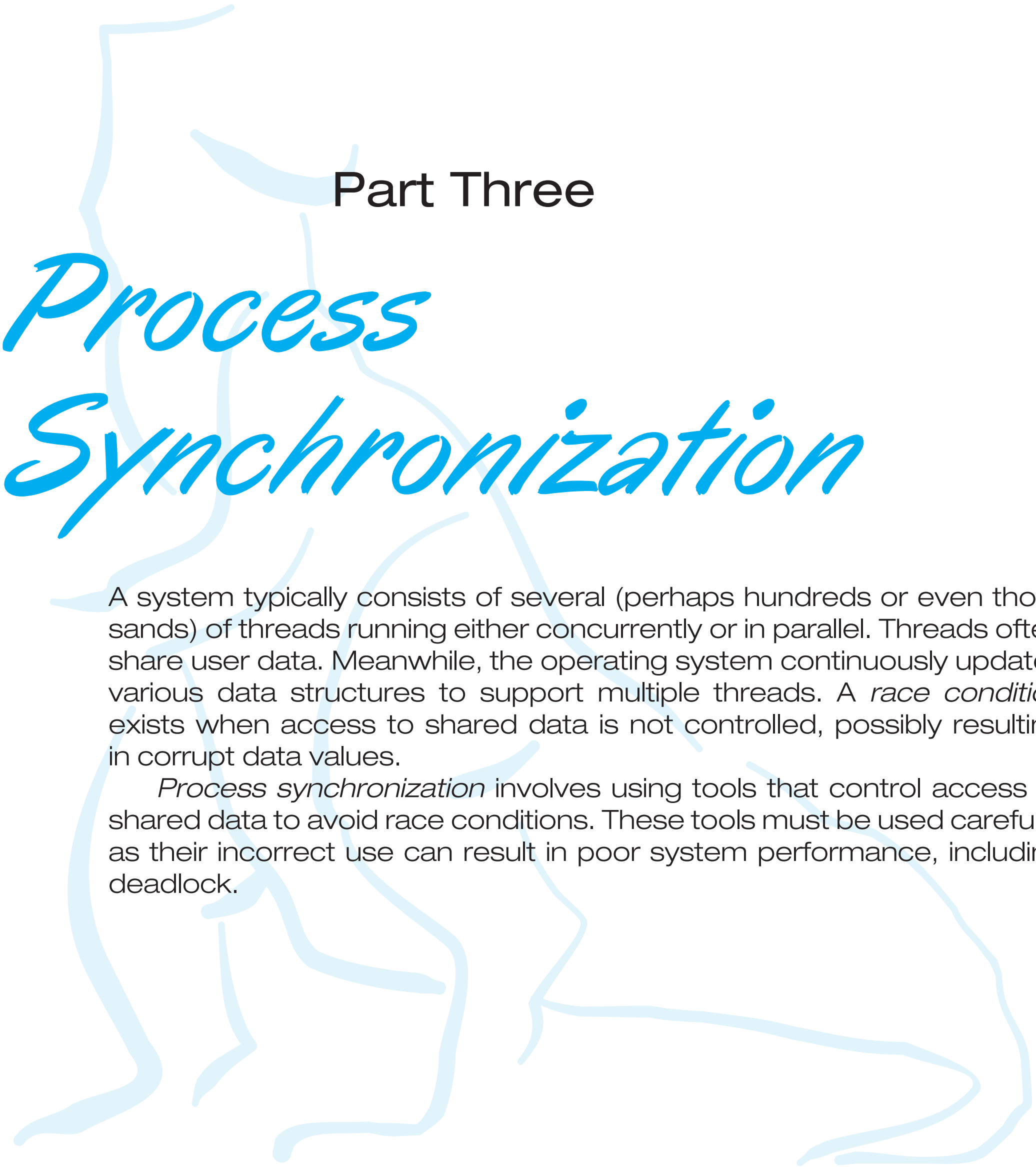
   ```
    int value = 0;
   __sync_fetch_and_add(&value,1);
   ```

   Refer to the Java API for details on how to use the `AtomicInteger` class for Java programs.

2. Calculate the average turnaround time, waiting time, and response time for each of the scheduling algorithms.

## Part Three

# Process Synchronization



A system typically consists of several (perhaps hundreds or even thousands) of threads running either concurrently or in parallel. Threads often share user data. Meanwhile, the operating system continuously updates various data structures to support multiple threads. A *race condition* exists when access to shared data is not controlled, possibly resulting in corrupt data values.

*Process synchronization* involves using tools that control access to shared data to avoid race conditions. These tools must be used carefully, as their incorrect use can result in poor system performance, including deadlock.

# Synchronization Tools

CHAPTER 6

A **cooperating process** is one that can affect or be affected by other processes executing in the system. Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through shared memory or message passing. Concurrent access to shared data may result in data inconsistency, however. In this chapter, we discuss various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

## CHAPTER OBJECTIVES

- Describe the critical-section problem and illustrate a race condition.
- Illustrate hardware solutions to the critical-section problem using memory barriers, compare-and-swap operations, and atomic variables.
- Demonstrate how mutex locks, semaphores, monitors, and condition variables can be used to solve the critical-section problem.
- Evaluate tools that solve the critical-section problem in low-, moderate-, and high-contention scenarios.

## 6.1 Background

We've already seen that processes can execute concurrently or in parallel. Section 3.2.2 introduced the role of process scheduling and described how the CPU scheduler switches rapidly between processes to provide concurrent execution. This means that one process may only partially complete execution before another process is scheduled. In fact, a process may be interrupted at any point in its instruction stream, and the processing core may be assigned to execute instructions of another process. Additionally, Section 4.2 introduced parallel execution, in which two instruction streams (representing different processes) execute simultaneously on separate processing cores. In this chapter, we explain how concurrent or parallel execution can contribute to issues involving the integrity of data shared by several processes.

Let's consider an example of how this can happen. In Chapter 3, we developed a model of a system consisting of cooperating sequential processes or threads, all running asynchronously and possibly sharing data. We illustrated this model with the producer−consumer problem, which is a representative paradigm of many operating system functions. Specifically, in Section 3.5, we described how a bounded buffer could be used to enable processes to share memory.

We now return to our consideration of the bounded buffer. As we pointed out, our original solution allowed at most `BUFFER_SIZE` − 1 items in the buffer at the same time. Suppose we want to modify the algorithm to remedy this deficiency. One possibility is to add an integer variable, `count`, initialized to 0. `count` is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer. The code for the producer process can be modified as follows:

```
while (true) {
     /* produce an item in next_produced */

     while (count == BUFFER_SIZE)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
     count++;
}
```

The code for the consumer process can be modified as follows:

```
while (true) {
     while (count == 0)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     count--;

     /* consume the item in next_consumed */
}
```

Although the producer and consumer routines shown above are correct separately, they may not function correctly when executed concurrently. As an illustration, suppose that the value of the variable `count` is currently 5 and that the producer and consumer processes concurrently execute the statements "`count++`" and "`count--`". Following the execution of these two statements, the value of the variable `count` may be 4, 5, or 6! The only correct result, though, is `count == 5`, which is generated correctly if the producer and consumer execute separately.

We can show that the value of `count` may be incorrect as follows. Note that the statement "`count++`" may be implemented in machine language (on a typical machine) as follows:

$$\begin{array}{l} \mathit{register}_1 = \texttt{count} \\ \mathit{register}_1 = \mathit{register}_1 + 1 \\ \texttt{count} = \mathit{register}_1 \end{array}$$

where $\mathit{register}_1$ is one of the local CPU registers. Similarly, the statement "`count--`" is implemented as follows:

$$\begin{array}{l} \mathit{register}_2 = \texttt{count} \\ \mathit{register}_2 = \mathit{register}_2 - 1 \\ \texttt{count} = \mathit{register}_2 \end{array}$$

where again $\mathit{register}_2$ is one of the local CPU registers. Even though $\mathit{register}_1$ and $\mathit{register}_2$ may be the same physical register, remember that the contents of this register will be saved and restored by the interrupt handler (Section 1.2.3).

The concurrent execution of "`count++`" and "`count--`" is equivalent to a sequential execution in which the lower-level statements presented previously are interleaved in some arbitrary order (but the order within each high-level statement is preserved). One such interleaving is the following:

| | | | | |
|---|---|---|---|---|
| $T_0$: | *producer* | execute | $\mathit{register}_1 = \texttt{count}$ | $\{\mathit{register}_1 = 5\}$ |
| $T_1$: | *producer* | execute | $\mathit{register}_1 = \mathit{register}_1 + 1$ | $\{\mathit{register}_1 = 6\}$ |
| $T_2$: | *consumer* | execute | $\mathit{register}_2 = \texttt{count}$ | $\{\mathit{register}_2 = 5\}$ |
| $T_3$: | *consumer* | execute | $\mathit{register}_2 = \mathit{register}_2 - 1$ | $\{\mathit{register}_2 = 4\}$ |
| $T_4$: | *producer* | execute | $\texttt{count} = \mathit{register}_1$ | $\{\mathit{count} = 6\}$ |
| $T_5$: | *consumer* | execute | $\texttt{count} = \mathit{register}_2$ | $\{\mathit{count} = 4\}$ |

Notice that we have arrived at the incorrect state "`count` == 4", indicating that four buffers are full, when, in fact, five buffers are full. If we reversed the order of the statements at $T_4$ and $T_5$, we would arrive at the incorrect state "`count` == 6".

We would arrive at this incorrect state because we allowed both processes to manipulate the variable `count` concurrently. A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**. To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable `count`. To make such a guarantee, we require that the processes be synchronized in some way.

Situations such as the one just described occur frequently in operating systems as different parts of the system manipulate resources. Furthermore, as we have emphasized in earlier chapters, the prominence of multicore systems has brought an increased emphasis on developing multithreaded applications. In such applications, several threads—which are quite possibly sharing data—are running in parallel on different processing cores. Clearly, we want any changes that result from such activities not to interfere with one another. Because of the importance of this issue, we devote a major portion of this chapter to **process synchronization** and **coordination** among cooperating processes.

## 6.2 The Critical-Section Problem

We begin our consideration of process synchronization by discussing the so-called critical-section problem. Consider a system consisting of $n$ processes $\{P_0, P_1, ..., P_{n-1}\}$. Each process has a segment of code, called a **critical section**, in which the process may be accessing — and updating — data that is shared with at least one other process. The important feature of the system is that, when one process is executing in its critical section, no other process is allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time. The ***critical-section problem*** is to design a protocol that the processes can use to synchronize their activity so as to cooperatively share data. Each process must request permission to enter its critical section. The section of code implementing this request is the **entry section**. The critical section may be followed by an **exit section**. The remaining code is the **remainder section**. The general structure of a typical process is shown in Figure 6.1. The entry section and exit section are enclosed in boxes to highlight these important segments of code.

A solution to the critical-section problem must satisfy the following three requirements:

1. **Mutual exclusion**. If process $P_i$ is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress**. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

```
while (true) {

    entry section

        critical section

    exit section

        remainder section

}
```

**Figure 6.1** General structure of a typical process.

3. **Bounded waiting**. There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

We assume that each process is executing at a nonzero speed. However, we can make no assumption concerning the relative speed of the $n$ processes.

At a given point in time, many kernel-mode processes may be active in the operating system. As a result, the code implementing an operating system (***kernel code***) is subject to several possible race conditions. Consider as an example a kernel data structure that maintains a list of all open files in the system. This list must be modified when a new file is opened or closed (adding the file to the list or removing it from the list). If two processes were to open files simultaneously, the separate updates to this list could result in a race condition.

Another example is illustrated in Figure 6.2. In this situation, two processes, $P_0$ and $P_1$, are creating child processes using the `fork()` system call. Recall from Section 3.3.1 that `fork()` returns the process identifier of the newly created process to the parent process. In this example, there is a race condition on the variable kernel variable `next_available_pid` which represents the value of the next available process identifier. Unless mutual exclusion is provided, it is possible the same process identifier number could be assigned to two separate processes.

Other kernel data structures that are prone to possible race conditions include structures for maintaining memory allocation, for maintaining process lists, and for interrupt handling. It is up to kernel developers to ensure that the operating system is free from such race conditions.

The critical-section problem could be solved simply in a single-core environment if we could prevent interrupts from occurring while a shared variable was being modified. In this way, we could be sure that the current sequence

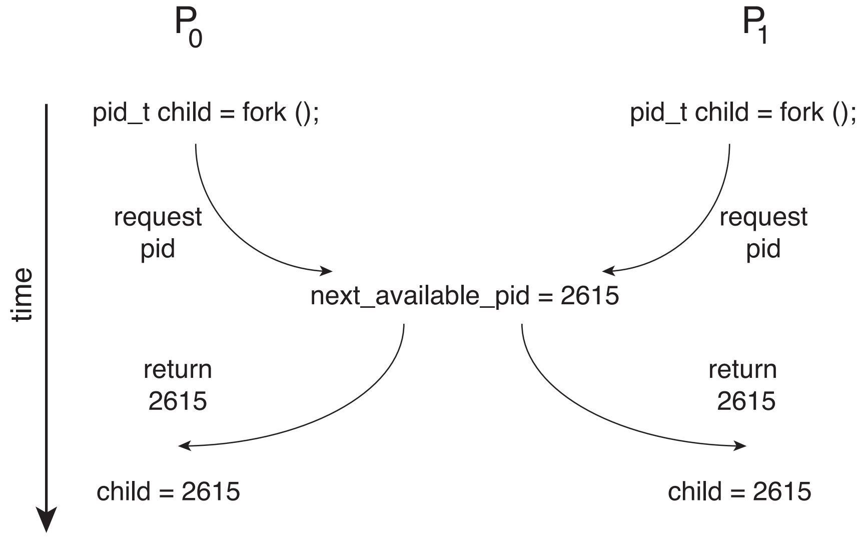


**Figure 6.2** Race condition when assigning a pid.

of instructions would be allowed to execute in order without preemption. No other instructions would be run, so no unexpected modifications could be made to the shared variable.

Unfortunately, this solution is not as feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time consuming, since the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases. Also consider the effect on a system's clock if the clock is kept updated by interrupts.

Two general approaches are used to handle critical sections in operating systems: **preemptive kernels** and **nonpreemptive kernels**. A preemptive kernel allows a process to be preempted while it is running in kernel mode. A nonpreemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

Obviously, a nonpreemptive kernel is essentially free from race conditions on kernel data structures, as only one process is active in the kernel at a time. We cannot say the same about preemptive kernels, so they must be carefully designed to ensure that shared kernel data are free from race conditions. Preemptive kernels are especially difficult to design for SMP architectures, since in these environments it is possible for two kernel-mode processes to run simultaneously on different CPU cores.

Why, then, would anyone favor a preemptive kernel over a nonpreemptive one? A preemptive kernel may be more responsive, since there is less risk that a kernel-mode process will run for an arbitrarily long period before relinquishing the processor to waiting processes. (Of course, this risk can also be minimized by designing kernel code that does not behave in this way.) Furthermore, a preemptive kernel is more suitable for real-time programming, as it will allow a real-time process to preempt a process currently running in the kernel.

## 6.3 Peterson's Solution

Next, we illustrate a classic software-based solution to the critical-section problem known as **Peterson's solution**. Because of the way modern computer architectures perform basic machine-language instructions, such as `load` and `store`, there are no guarantees that Peterson's solution will work correctly on such architectures. However, we present the solution because it provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting.

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered $P_0$ and $P_1$. For convenience, when presenting $P_i$, we use $P_j$ to denote the other process; that is, `j` equals 1 − `i`.

Peterson's solution requires the two processes to share two data items:

```
int turn;
boolean flag[2];
```

```
while (true) {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j)
        ;

        /* critical section */

    flag[i] = false;

        /*remainder section */
}
```

**Figure 6.3** The structure of process $P_i$ in Peterson's solution.

The variable `turn` indicates whose turn it is to enter its critical section. That is, if `turn == i`, then process $P_i$ is allowed to execute in its critical section. The `flag` array is used to indicate if a process is ready to enter its critical section. For example, if `flag[i]` is `true`, $P_i$ is ready to enter its critical section. With an explanation of these data structures complete, we are now ready to describe the algorithm shown in Figure 6.3.

To enter the critical section, process $P_i$ first sets `flag[i]` to be `true` and then sets `turn` to the value `j`, thereby asserting that if the other process wishes to enter the critical section, it can do so. If both processes try to enter at the same time, `turn` will be set to both `i` and `j` at roughly the same time. Only one of these assignments will last; the other will occur but will be overwritten immediately. The eventual value of `turn` determines which of the two processes is allowed to enter its critical section first.

We now prove that this solution is correct. We need to show that:

1. Mutual exclusion is preserved.
2. The progress requirement is satisfied.
3. The bounded-waiting requirement is met.

To prove property 1, we note that each $P_i$ enters its critical section only if either `flag[j] == false` or `turn == i`. Also note that, if both processes can be executing in their critical sections at the same time, then `flag[0] == flag[1] == true`. These two observations imply that $P_0$ and $P_1$ could not have successfully executed their `while` statements at about the same time, since the value of `turn` can be either 0 or 1 but cannot be both. Hence, one of the processes—say, $P_j$—must have successfully executed the while statement, whereas $P_i$ had to execute at least one additional statement ("`turn == j`"). However, at that time, `flag[j] == true` and `turn == j`, and this condition will persist as long as $P_j$ is in its critical section; as a result, mutual exclusion is preserved.

To prove properties 2 and 3, we note that a process $P_i$ can be prevented from entering the critical section only if it is stuck in the `while` loop with the condition `flag[j] == true` and `turn == j`; this loop is the only one possible. If $P_j$ is not ready to enter the critical section, then `flag[j] == false`, and $P_i$ can enter its critical section. If $P_j$ has set `flag[j]` to `true` and is also executing in its `while` statement, then either `turn == i` or `turn == j`. If `turn == i`, then $P_i$ will enter the critical section. If `turn == j`, then $P_j$ will enter the critical section. However, once $P_j$ exits its critical section, it will reset `flag[j]` to `false`, allowing $P_i$ to enter its critical section. If $P_j$ resets `flag[j]` to `true`, it must also set `turn` to `i`. Thus, since $P_i$ does not change the value of the variable `turn` while executing the `while` statement, $P_i$ will enter the critical section (progress) after at most one entry by $P_j$ (bounded waiting).

As mentioned at the beginning of this section, Peterson's solution is not guaranteed to work on modern computer architectures for the primary reason that, to improve system performance, processors and/or compilers may reorder read and write operations that have no dependencies. For a single-threaded application, this reordering is immaterial as far as program correctness is concerned, as the final values are consistent with what is expected. (This is similar to balancing a checkbook—the actual order in which credit and debit operations are performed is unimportant, because the final balance will still be the same.) But for a multithreaded application with shared data, the reordering of instructions may render inconsistent or unexpected results.

As an example, consider the following data that are shared between two threads:

```
boolean flag = false;
int x = 0;
```

where Thread 1 performs the statements

```
while (!flag)
   ;
print x;
```

and Thread 2 performs

```
x = 100;
flag = true;
```

The expected behavior is, of course, that Thread 1 outputs the value 100 for variable `x`. However, as there are no data dependencies between the variables `flag` and `x`, it is possible that a processor may reorder the instructions for Thread 2 so that `flag` is assigned `true` before assignment of `x = 100`. In this situation, it is possible that Thread 1 would output 0 for variable `x`. Less obvious is that the processor may also reorder the statements issued by Thread 1 and load the variable `x` before loading the value of `flag`. If this were to occur, Thread 1 would output 0 for variable `x` even if the instructions issued by Thread 2 were not reordered.

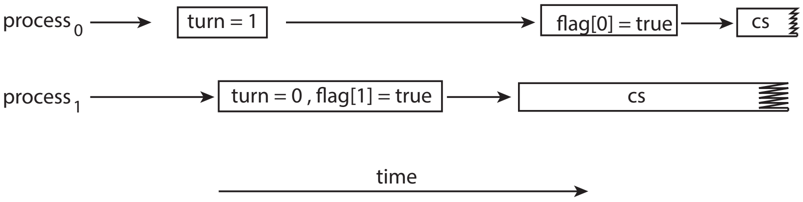


**Figure 6.4** The effects of instruction reordering in Peterson's solution.

How does this affect Peterson's solution? Consider what happens if the assignments of the first two statements that appear in the entry section of Peterson's solution in Figure 6.3 are reordered; it is possible that both threads may be active in their critical sections at the same time, as shown in Figure 6.4.

As you will see in the following sections, the only way to preserve mutual exclusion is by using proper synchronization tools. Our discussion of these tools begins with primitive support in hardware and proceeds through abstract, high-level, software-based APIs available to both kernel developers and application programmers.

## 6.4 Hardware Support for Synchronization

We have just described one software-based solution to the critical-section problem. (We refer to it as a ***software-based*** solution because the algorithm involves no special support from the operating system or specific hardware instructions to ensure mutual exclusion.) However, as discussed, software-based solutions are not guaranteed to work on modern computer architectures. In this section, we present three hardware instructions that provide support for solving the critical-section problem. These primitive operations can be used directly as synchronization tools, or they can be used to form the foundation of more abstract synchronization mechanisms.

### 6.4.1 Memory Barriers

In Section 6.3, we saw that a system may reorder instructions, a policy that can lead to unreliable data states. How a computer architecture determines what memory guarantees it will provide to an application program is known as its **memory model**. In general, a memory model falls into one of two categories:

1. **Strongly ordered**, where a memory modification on one processor is immediately visible to all other processors.
2. **Weakly ordered**, where modifications to memory on one processor may not be immediately visible to other processors.

Memory models vary by processor type, so kernel developers cannot make any assumptions regarding the visibility of modifications to memory on a shared-memory multiprocessor. To address this issue, computer architectures provide instructions that can *force* any changes in memory to be propagated to all other processors, thereby ensuring that memory modifications are visible to threads running on other processors. Such instructions are known as **memory barriers** or **memory fences**. When a memory barrier instruction is performed, the system ensures that all loads and stores are completed before any subsequent load or store operations are performed. Therefore, even if instructions were reordered, the memory barrier ensures that the store operations are completed in memory and visible to other processors before future load or store operations are performed.

Let's return to our most recent example, in which reordering of instructions could have resulted in the wrong output, and use a memory barrier to ensure that we obtain the expected output.

If we add a memory barrier operation to Thread 1

```
while (!flag)
  memory_barrier();
print x;
```

we guarantee that the value of `flag` is loaded before the value of `x`.

Similarly, if we place a memory barrier between the assignments performed by Thread 2

```
x = 100;
memory_barrier();
flag = true;
```

we ensure that the assignment to `x` occurs before the assignment to `flag`.

With respect to Peterson's solution, we could place a memory barrier between the first two assignment statements in the entry section to avoid the reordering of operations shown in Figure 6.4. Note that memory barriers are considered very low-level operations and are typically only used by kernel developers when writing specialized code that ensures mutual exclusion.

### 6.4.2 Hardware Instructions

Many modern computer systems provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words **atomically**—that is, as one uninterruptible unit. We can use these special instructions to solve the critical-section problem in a relatively simple manner. Rather than discussing one specific instruction for one specific machine, we abstract the main concepts behind these types of instructions by describing the `test_and_set()` and `compare_and_swap()` instructions.

```
boolean test_and_set(boolean *target) {
  boolean rv = *target;
  *target = true;

  return rv;
}
```

**Figure 6.5** The definition of the atomic `test_and_set()` instruction.

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

**Figure 6.6** Mutual-exclusion implementation with `test_and_set()`.

The `test_and_set()` instruction can be defined as shown in Figure 6.5. The important characteristic of this instruction is that it is executed atomically. Thus, if two `test_and_set()` instructions are executed simultaneously (each on a different core), they will be executed sequentially in some arbitrary order. If the machine supports the `test_and_set()` instruction, then we can implement mutual exclusion by declaring a boolean variable `lock`, initialized to `false`. The structure of process $P_i$ is shown in Figure 6.6.

The `compare_and_swap()` instruction (CAS), just like the `test_and_set()` instruction, operates on two words atomically, but uses a different mechanism that is based on swapping the content of two words.

The CAS instruction operates on three operands and is defined in Figure 6.7. The operand `value` is set to `new_value` only if the expression `(*value == expected)` is true. Regardless, CAS always returns the original value of the variable `value`. The important characteristic of this instruction is that it is executed atomically. Thus, if two CAS instructions are executed simultaneously (each on a different core), they will be executed sequentially in some arbitrary order.

Mutual exclusion using CAS can be provided as follows: A global variable (`lock`) is declared and is initialized to 0. The first process that invokes `compare_and_swap()` will set `lock` to 1. It will then enter its critical section,

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

**Figure 6.7** The definition of the atomic `compare_and_swap()` instruction.

```
while (true) {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

      /* critical section */

   lock = 0;

      /* remainder section */
}
```

**Figure 6.8** Mutual exclusion with the `compare_and_swap()` instruction.

because the original value of `lock` was equal to the expected value of 0. Subsequent calls to `compare_and_swap()` will not succeed, because `lock` now is not equal to the expected value of 0. When a process exits its critical section, it sets `lock` back to 0, which allows another process to enter its critical section. The structure of process $P_i$ is shown in Figure 6.8.

Although this algorithm satisfies the mutual-exclusion requirement, it does not satisfy the bounded-waiting requirement. In Figure 6.9, we present

```
while (true) {
   waiting[i] = true;
   key = 1;
   while (waiting[i] && key == 1)
      key = compare_and_swap(&lock,0,1);
   waiting[i] = false;

      /* critical section */

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = 0;
   else
      waiting[j] = false;

      /* remainder section */
}
```

**Figure 6.9** Bounded-waiting mutual exclusion with `compare_and_swap()`.

### *MAKING COMPARE-AND-SWAP ATOMIC*

On Intel x86 architectures, the assembly language statement `cmpxchg` is used to implement the `compare_and_swap()` instruction. To enforce atomic execution, the `lock` prefix is used to lock the bus while the destination operand is being updated. The general form of this instruction appears as:

```
lock cmpxchg <destination operand>, <source operand>
```

another algorithm using the `compare_and_swap()` instruction that satisfies all the critical-section requirements. The common data structures are

```
boolean waiting[n];
int lock;
```

The elements in the `waiting` array are initialized to `false`, and `lock` is initialized to 0. To prove that the mutual-exclusion requirement is met, we note that process $P_i$ can enter its critical section only if either `waiting[i] == false` or `key == 0`. The value of `key` can become `0` only if the `compare_and_swap()` is executed. The first process to execute the `compare_and_swap()` will find `key == 0`; all others must wait. The variable `waiting[i]` can become `false` only if another process leaves its critical section; only one `waiting[i]` is set to `false`, maintaining the mutual-exclusion requirement.

To prove that the progress requirement is met, we note that the arguments presented for mutual exclusion also apply here, since a process exiting the critical section either sets `lock` to `0` or sets `waiting[j]` to `false`. Both allow a process that is waiting to enter its critical section to proceed.

To prove that the bounded-waiting requirement is met, we note that, when a process leaves its critical section, it scans the array `waiting` in the cyclic ordering ($i + 1, i + 2, ..., n - 1, 0, ..., i - 1$). It designates the first process in this ordering that is in the entry section (`waiting[j] == true`) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within $n - 1$ turns.

Details describing the implementation of the atomic `test_and_set()` and `compare_and_swap()` instructions are discussed more fully in books on computer architecture.

### 6.4.3 Atomic Variables

Typically, the `compare_and_swap()` instruction is not used directly to provide mutual exclusion. Rather, it is used as a basic building block for constructing other tools that solve the critical-section problem. One such tool is an **atomic variable**, which provides atomic operations on basic data types such as integers and booleans. We know from Section 6.1 that incrementing or decrementing an integer value may produce a race condition. Atomic variables can be used in to ensure mutual exclusion in situations where there may be a data race on a single variable while it is being updated, as when a counter is incremented.

Most systems that support atomic variables provide special atomic data types as well as functions for accessing and manipulating atomic variables.

These functions are often implemented using `compare_and_swap()` operations. As an example, the following increments the atomic integer `sequence`:

```
increment(&sequence);
```

where the `increment()` function is implemented using the CAS instruction:

```
void increment(atomic_int *v)
{
   int temp;

   do {
      temp = *v;
   }
   while (temp != compare_and_swap(v, temp, temp+1));
}
```

It is important to note that although atomic variables provide atomic updates, they do not entirely solve race conditions in all circumstances. For example, in the bounded-buffer problem described in Section 6.1, we could use an atomic integer for `count`. This would ensure that the updates to `count` were atomic. However, the producer and consumer processes also have `while` loops whose condition depends on the value of `count`. Consider a situation in which the buffer is currently empty and two consumers are looping while waiting for `count` > 0. If a producer entered one item in the buffer, both consumers could exit their `while` loops (as `count` would no longer be equal to 0) and proceed to consume, even though the value of `count` was only set to 1.

Atomic variables are commonly used in operating systems as well as concurrent applications, although their use is often limited to single updates of shared data such as counters and sequence generators. In the following sections, we explore more robust tools that address race conditions in more generalized situations.

## 6.5 Mutex Locks

The hardware-based solutions to the critical-section problem presented in Section 6.4 are complicated as well as generally inaccessible to application programmers. Instead, operating-system designers build higher-level software tools to solve the critical-section problem. The simplest of these tools is the **mutex lock**. (In fact, the term ***mutex*** is short for ***mut***ual ***ex***clusion.) We use the mutex lock to protect critical sections and thus prevent race conditions. That is, a process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section. The `acquire()` function acquires the lock, and the `release()` function releases the lock, as illustrated in Figure 6.10.

A mutex lock has a boolean variable `available` whose value indicates if the lock is available or not. If the lock is available, a call to `acquire()` succeeds, and the lock is then considered unavailable. A process that attempts to acquire an unavailable lock is blocked until the lock is released.

```
while (true) {

      acquire lock

            critical section

      release lock

            remainder section

}
```

Figure 6.10 Solution to the critical-section problem using mutex locks.

The definition of `acquire()` is as follows:

```
acquire() {
   while (!available)
      ; /* busy wait */
   available = false;
}
```

The definition of `release()` is as follows:

```
release() {
  available = true;
}
```

Calls to either `acquire()` or `release()` must be performed atomically. Thus, mutex locks can be implemented using the CAS operation described in Section 6.4, and we leave the description of this technique as an exercise.

### *LOCK CONTENTION*

Locks are either contended or uncontended. A lock is considered **contended** if a thread blocks while trying to acquire the lock. If a lock is available when a thread attempts to acquire it, the lock is considered **uncontended**. Contended locks can experience either ***high contention*** (a relatively large number of threads attempting to acquire the lock) or ***low contention*** (a relatively small number of threads attempting to acquire the lock.) Unsurprisingly, highly contended locks tend to decrease overall performance of concurrent applications.

### *WHAT IS MEANT BY "SHORT DURATION"?*

Spinlocks are often identified as the locking mechanism of choice on multiprocessor systems when the lock is to be held for a short duration. But what exactly constitutes a *short duration*? Given that waiting on a lock requires two context switches—a context switch to move the thread to the waiting state and a second context switch to restore the waiting thread once the lock becomes available—the general rule is to use a spinlock if the lock will be held for a duration of less than two context switches.

The main disadvantage of the implementation given here is that it requires **busy waiting**. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to `acquire()`. This continual looping is clearly a problem in a real multiprogramming system, where a single CPU core is shared among many processes. Busy waiting also wastes CPU cycles that some other process might be able to use productively. (In Section 6.6, we examine a strategy that avoids busy waiting by temporarily putting the waiting process to sleep and then awakening it once the lock becomes available.)

The type of mutex lock we have been describing is also called a **spinlock** because the process "spins" while waiting for the lock to become available. (We see the same issue with the code examples illustrating the `compare_and_swap()` instruction.) Spinlocks do have an advantage, however, in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. In certain circumstances on multicore systems, spinlocks are in fact the preferable choice for locking. If a lock is to be held for a short duration, one thread can "spin" on one processing core while another thread performs its critical section on another core. On modern multicore computing systems, spinlocks are widely used in many operating systems.

In Chapter 7 we examine how mutex locks can be used to solve classical synchronization problems. We also discuss how mutex locks and spinlocks are used in several operating systems, as well as in Pthreads.

## 6.6 Semaphores

Mutex locks, as we mentioned earlier, are generally considered the simplest of synchronization tools. In this section, we examine a more robust tool that can behave similarly to a mutex lock but can also provide more sophisticated ways for processes to synchronize their activities.

A **semaphore** `S` is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: `wait()` and `signal()`. Semaphores were introduced by the Dutch computer scientist Edsger Dijkstra, and such, the `wait()` operation was originally termed `P` (from the Dutch

***proberen,*** "to test"); `signal()` was originally called `V` (from ***verhogen,*** "to increment"). The definition of `wait()` is as follows:

```
wait(S) {
   while (S <= 0)
      ; // busy wait
   S--;
}
```

The definition of `signal()` is as follows:

```
signal(S) {
   S++;
}
```

All modifications to the integer value of the semaphore in the `wait()` and `signal()` operations must be executed atomically. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of `wait(S)`, the testing of the integer value of `S` (`S` $\leq 0$), as well as its possible modification (`S--`), must be executed without interruption. We shall see how these operations can be implemented in Section 6.6.2. First, let's see how semaphores can be used.

### 6.6.1 Semaphore Usage

Operating systems often distinguish between counting and binary semaphores. The value of a **counting semaphore** can range over an unrestricted domain. The value of a **binary semaphore** can range only between 0 and 1. Thus, binary semaphores behave similarly to mutex locks. In fact, on systems that do not provide mutex locks, binary semaphores can be used instead for providing mutual exclusion.

Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a `wait()` operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a `signal()` operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

We can also use semaphores to solve various synchronization problems. For example, consider two concurrently running processes: $P_1$ with a statement $S_1$ and $P_2$ with a statement $S_2$. Suppose we require that $S_2$ be executed only after $S_1$ has completed. We can implement this scheme readily by letting $P_1$ and $P_2$ share a common semaphore `synch`, initialized to 0. In process $P_1$, we insert the statements

```
S1;
signal(synch);
```

In process $P_2$, we insert the statements

```
wait(synch);
S2;
```

Because `synch` is initialized to 0, $P_2$ will execute $S_2$ only after $P_1$ has invoked `signal(synch)`, which is after statement $S_1$ has been executed.

### 6.6.2 Semaphore Implementation

Recall that the implementation of mutex locks discussed in Section 6.5 suffers from busy waiting. The definitions of the `wait()` and `signal()` semaphore operations just described present the same problem. To overcome this problem, we can modify the definition of the `wait()` and `signal()` operations as follows: When a process executes the `wait()` operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process can suspend itself. The suspend operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

A process that is suspended, waiting on a semaphore `S`, should be restarted when some other process executes a `signal()` operation. The process is restarted by a `wakeup()` operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue. (The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU-scheduling algorithm.)

To implement semaphores under this definition, we define a semaphore as follows:

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

Each semaphore has an integer `value` and a list of processes `list`. When a process must wait on a semaphore, it is added to the list of processes. A `signal()` operation removes one process from the list of waiting processes and awakens that process.

Now, the `wait()` semaphore operation can be defined as

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                sleep();
        }
}
```

and the `signal()` semaphore operation can be defined as

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
              remove a process P from S->list;
              wakeup(P);
        }
}
```

The `sleep()` operation suspends the process that invokes it. The `wakeup(P)` operation resumes the execution of a suspended process `P`. These two operations are provided by the operating system as basic system calls.

Note that in this implementation, semaphore values may be negative, whereas semaphore values are never negative under the classical definition of semaphores with busy waiting. If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore. This fact results from switching the order of the decrement and the test in the implementation of the `wait()` operation.

The list of waiting processes can be easily implemented by a link field in each process control block (PCB). Each semaphore contains an integer value and a pointer to a list of PCBs. One way to add and remove processes from the list so as to ensure bounded waiting is to use a FIFO queue, where the semaphore contains both head and tail pointers to the queue. In general, however, the list can use any queuing strategy. Correct usage of semaphores does not depend on a particular queuing strategy for the semaphore lists.

As mentioned, it is critical that semaphore operations be executed atomically. We must guarantee that no two processes can execute `wait()` and `signal()` operations on the same semaphore at the same time. This is a critical-section problem, and in a single-processor environment, we can solve it by simply inhibiting interrupts during the time the `wait()` and `signal()` operations are executing. This scheme works in a single-processor environment because, once interrupts are inhibited, instructions from different processes cannot be interleaved. Only the currently running process executes until interrupts are reenabled and the scheduler can regain control.

In a multicore environment, interrupts must be disabled on every processing core. Otherwise, instructions from different processes (running on different cores) may be interleaved in some arbitrary way. Disabling interrupts on every core can be a difficult task and can seriously diminish performance. Therefore, SMP systems must provide alternative techniques—such as `compare_and_swap()` or spinlocks—to ensure that `wait()` and `signal()` are performed atomically.

It is important to admit that we have not completely eliminated busy waiting with this definition of the `wait()` and `signal()` operations. Rather, we have moved busy waiting from the entry section to the critical sections of application programs. Furthermore, we have limited busy waiting to the critical sections of the `wait()` and `signal()` operations, and these sections are short (if properly coded, they should be no more than about ten instructions). Thus, the critical section is almost never occupied, and busy waiting occurs rarely, and then for only a short time. An entirely different situation exists with application programs whose critical sections may be long (minutes or even hours) or may almost always be occupied. In such cases, busy waiting is extremely inefficient.

## 6.7 Monitors

Although semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in timing errors that are difficult to detect, since these errors happen only if particular execution sequences take place, and these sequences do not always occur.

We have seen an example of such errors in the use of a count in our solution to the producer–consumer problem (Section 6.1). In that example, the timing problem happened only rarely, and even then the `count` value appeared to be reasonable—off by only 1. Nevertheless, the solution is obviously not an acceptable one. It is for this reason that mutex locks and semaphores were introduced in the first place.

Unfortunately, such timing errors can still occur when either mutex locks or semaphores are used. To illustrate how, we review the semaphore solution to the critical-section problem. All processes share a binary semaphore variable `mutex`, which is initialized to 1. Each process must execute `wait(mutex)` before entering the critical section and `signal(mutex)` afterward. If this sequence is not observed, two processes may be in their critical sections simultaneously. Next, we list several difficulties that may result. Note that these difficulties will arise even if a ***single*** process is not well behaved. This situation may be caused by an honest programming error or an uncooperative programmer.

- Suppose that a program interchanges the order in which the `wait()` and `signal()` operations on the semaphore `mutex` are executed, resulting in the following execution:

  ```
  signal(mutex);
      ...
    critical section
      ...
  wait(mutex);
  ```

  In this situation, several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement. This error may be discovered only if several processes are simultaneously active in their critical sections. Note that this situation may not always be reproducible.

- Suppose that a program replaces `signal(mutex)` with `wait(mutex)`. That is, it executes

  ```
  wait(mutex);
      ...
    critical section
      ...
  wait(mutex);
  ```

In this case, the process will permanently block on the second call to `wait()`, as the semaphore is now unavailable.

- Suppose that a process omits the `wait(mutex)`, or the `signal(mutex)`, or both. In this case, either mutual exclusion is violated or the process will permanently block.

These examples illustrate that various types of errors can be generated easily when programmers use semaphores or mutex locks incorrectly to solve the critical-section problem. One strategy for dealing with such errors is to incorporate simple synchronization tools as high-level language constructs. In this section, we describe one fundamental high-level synchronization construct—the **monitor** type.

### 6.7.1 Monitor Usage

An **abstract data type**—or **ADT**—encapsulates data with a set of functions to operate on that data that are independent of any specific implementation of the ADT. A ***monitor type*** is an ADT that includes a set of programmer-defined operations that are provided with mutual exclusion within the monitor. The monitor type also declares the variables whose values define the state of an

```
monitor monitor name
{
   /* shared variable declarations */

   function P1 ( . . . ) {
      . . .
   }

   function P2 ( . . . ) {
      . . .
   }

         .
         .
         .
   function Pn ( . . . ) {
      . . .
   }

   initialization_code ( . . . ) {
      . . .
   }
}
```

Figure 6.11 Pseudocode syntax of a monitor.

instance of that type, along with the bodies of functions that operate on those variables. The syntax of a monitor type is shown in Figure 6.11. The representation of a monitor type cannot be used directly by the various processes. Thus, a function defined within a monitor can access only those variables declared locally within the monitor and its formal parameters. Similarly, the local variables of a monitor can be accessed by only the local functions.

The monitor construct ensures that only one process at a time is active within the monitor. Consequently, the programmer does not need to code this synchronization constraint explicitly (Figure 6.12). However, the monitor construct, as defined so far, is not sufficiently powerful for modeling some synchronization schemes. For this purpose, we need to define additional synchronization mechanisms. These mechanisms are provided by the `condition` construct. A programmer who needs to write a tailor-made synchronization scheme can define one or more variables of type *condition*:

```
condition x, y;
```

The only operations that can be invoked on a condition variable are `wait()` and `signal()`. The operation

```
x.wait();
```

means that the process invoking this operation is suspended until another process invokes

```
x.signal();
```

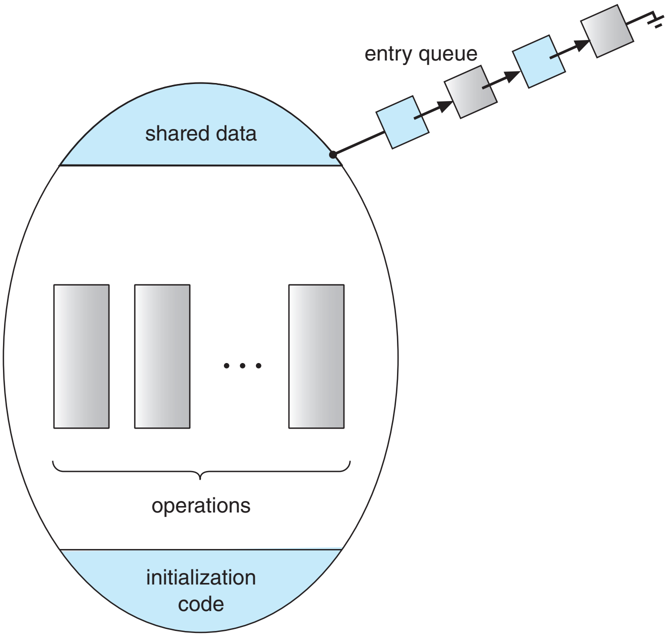


**Figure 6.12** Schematic view of a monitor.

The `x.signal()` operation resumes exactly one suspended process. If no process is suspended, then the `signal()` operation has no effect; that is, the state of `x` is the same as if the operation had never been executed (Figure 6.13). Contrast this operation with the `signal()` operation associated with semaphores, which always affects the state of the semaphore.

Now suppose that, when the `x.signal()` operation is invoked by a process *P*, there exists a suspended process *Q* associated with condition `x`. Clearly, if the suspended process *Q* is allowed to resume its execution, the signaling process *P* must wait. Otherwise, both *P* and *Q* would be active simultaneously within the monitor. Note, however, that conceptually both processes can continue with their execution. Two possibilities exist:

1. **Signal and wait**. *P* either waits until *Q* leaves the monitor or waits for another condition.
2. **Signal and continue**. *Q* either waits until *P* leaves the monitor or waits for another condition.

There are reasonable arguments in favor of adopting either option. On the one hand, since *P* was already executing in the monitor, the ***signal-and-continue*** method seems more reasonable. On the other, if we allow thread *P* to continue, then by the time *Q* is resumed, the logical condition for which *Q* was waiting may no longer hold. A compromise between these two choices exists as well: when thread *P* executes the signal operation, it immediately leaves the monitor. Hence, *Q* is immediately resumed.

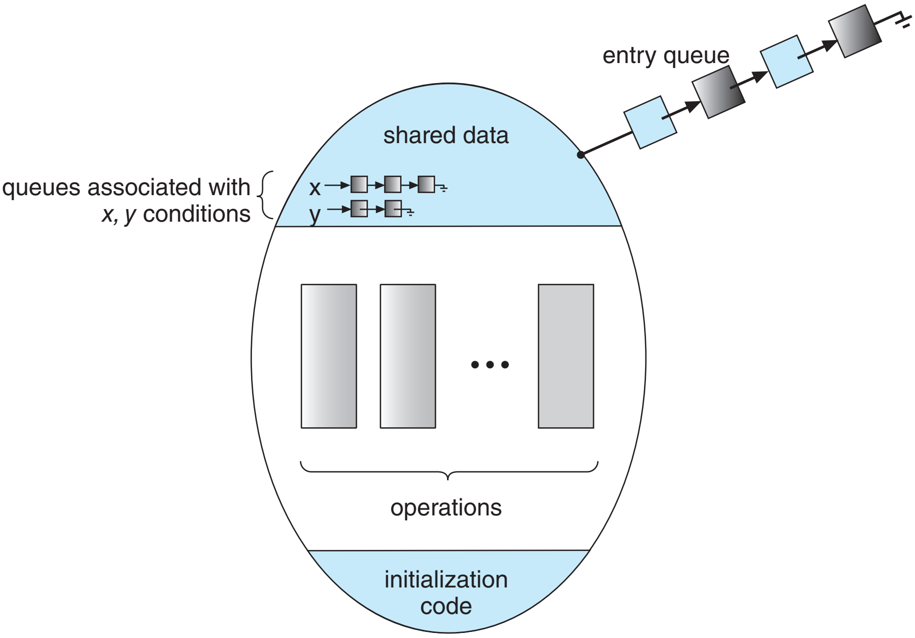


**Figure 6.13** Monitor with condition variables.

Many programming languages have incorporated the idea of the monitor as described in this section, including Java and C#. Other languages—such as Erlang—provide concurrency support using a similar mechanism.

### 6.7.2 Implementing a Monitor Using Semaphores

We now consider a possible implementation of the monitor mechanism using semaphores. For each monitor, a binary semaphore `mutex` (initialized to 1) is provided to ensure mutual exclusion. A process must execute `wait(mutex)` before entering the monitor and must execute `signal(mutex)` after leaving the monitor.

We will use the signal-and-wait scheme in our implementation. Since a signaling process must wait until the resumed process either leaves or waits, an additional binary semaphore, `next`, is introduced, initialized to 0. The signaling processes can use `next` to suspend themselves. An integer variable `next_count` is also provided to count the number of processes suspended on `next`. Thus, each external function `F` is replaced by

```
wait(mutex);
    ...
   body of F
    ...
if (next_count > 0)
   signal(next);
else
   signal(mutex);
```

Mutual exclusion within a monitor is ensured.

We can now describe how condition variables are implemented as well. For each condition `x`, we introduce a binary semaphore `x_sem` and an integer variable `x_count`, both initialized to 0. The operation `x.wait()` can now be implemented as

```
x_count++;
if (next_count > 0)
   signal(next);
else
   signal(mutex);
wait(x_sem);
x_count--;
```

The operation `x.signal()` can be implemented as

```
if (x_count > 0) {
   next_count++;
   signal(x_sem);
   wait(next);
   next_count--;
}
```

This implementation is applicable to the definitions of monitors given by both Hoare and Brinch-Hansen (see the bibliographical notes at the end of the chapter). In some cases, however, the generality of the implementation is

```
monitor ResourceAllocator
{
   boolean busy;
   condition x;

   void acquire(int time) {
      if (busy)
         x.wait(time);
      busy = true;
   }

   void release() {
      busy = false;
      x.signal();
   }

   initialization_code() {
      busy = false;
   }
}
```

**Figure 6.14** A monitor to allocate a single resource.

unnecessary, and a significant improvement in efficiency is possible. We leave this problem to you in Exercise 6.27.

### 6.7.3 Resuming Processes within a Monitor

We turn now to the subject of process-resumption order within a monitor. If several processes are suspended on condition `x`, and an `x.signal()` operation is executed by some process, then how do we determine which of the suspended processes should be resumed next? One simple solution is to use a first-come, first-served (FCFS) ordering, so that the process that has been waiting the longest is resumed first. In many circumstances, however, such a simple scheduling scheme is not adequate. In these circumstances, the **conditional-wait** construct can be used. This construct has the form

```
x.wait(c);
```

where `c` is an integer expression that is evaluated when the `wait()` operation is executed. The value of `c`, which is called a **priority number**, is then stored with the name of the process that is suspended. When `x.signal()` is executed, the process with the smallest priority number is resumed next.

To illustrate this new mechanism, consider the `ResourceAllocator` monitor shown in Figure 6.14, which controls the allocation of a single resource among competing processes. Each process, when requesting an allocation of this resource, specifies the maximum time it plans to use the resource. The monitor allocates the resource to the process that has the shortest time-allocation request. A process that needs to access the resource in question must observe the following sequence:

```
R.acquire(t);
     ...
   access the resource;
     ...
R.release();
```

where `R` is an instance of type `ResourceAllocator`.

Unfortunately, the monitor concept cannot guarantee that the preceding access sequence will be observed. In particular, the following problems can occur:

- A process might access a resource without first gaining access permission to the resource.
- A process might never release a resource once it has been granted access to the resource.
- A process might attempt to release a resource that it never requested.
- A process might request the same resource twice (without first releasing the resource).

The same difficulties are encountered with the use of semaphores, and these difficulties are similar in nature to those that encouraged us to develop the monitor constructs in the first place. Previously, we had to worry about the correct use of semaphores. Now, we have to worry about the correct use of higher-level programmer-defined operations, with which the compiler can no longer assist us.

One possible solution to the current problem is to include the resource-access operations within the `ResourceAllocator` monitor. However, using this solution will mean that scheduling is done according to the built-in monitor-scheduling algorithm rather than the one we have coded.

To ensure that the processes observe the appropriate sequences, we must inspect all the programs that make use of the `ResourceAllocator` monitor and its managed resource. We must check two conditions to establish the correctness of this system. First, user processes must always make their calls on the monitor in a correct sequence. Second, we must be sure that an uncooperative process does not simply ignore the mutual-exclusion gateway provided by the monitor and try to access the shared resource directly, without using the access protocols. Only if these two conditions can be ensured can we guarantee that no time-dependent errors will occur and that the scheduling algorithm will not be defeated.

Although this inspection may be possible for a small, static system, it is not reasonable for a large system or a dynamic system. This access-control problem can be solved only through the use of the additional mechanisms that are described in Chapter 17.

## 6.8 Liveness

One consequence of using synchronization tools to coordinate access to critical sections is the possibility that a process attempting to enter its critical section will wait indefinitely. Recall that in Section 6.2, we outlined three criteria that solutions to the critical-section problem must satisfy. Indefinite waiting violates two of these—the progress and bounded-waiting criteria.

**Liveness** refers to a set of properties that a system must satisfy to ensure that processes make progress during their execution life cycle. A process waiting indefinitely under the circumstances just described is an example of a "liveness failure."

There are many different forms of liveness failure; however, all are generally characterized by poor performance and responsiveness. A very simple example of a liveness failure is an infinite loop. A busy wait loop presents the *possibility* of a liveness failure, especially if a process may loop an arbitrarily long period of time. Efforts at providing mutual exclusion using tools such as mutex locks and semaphores can often lead to such failures in concurrent programming. In this section, we explore two situations that can lead to liveness failures.

### 6.8.1 Deadlock

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a `signal()` operation. When such a state is reached, these processes are said to be **deadlocked**.

To illustrate this, consider a system consisting of two processes, $P_0$ and $P_1$, each accessing two semaphores, `S` and `Q`, set to the value 1:

```
     P0             P1

wait(S);        wait(Q);
wait(Q);        wait(S);
   .               .
   .               .
   .               .
signal(S);      signal(Q);
signal(Q);      signal(S);
```

Suppose that $P_0$ executes `wait(S)` and then $P_1$ executes `wait(Q)`. When $P_0$ executes `wait(Q)`, it must wait until $P_1$ executes `signal(Q)`. Similarly, when $P_1$ executes `wait(S)`, it must wait until $P_0$ executes `signal(S)`. Since these `signal()` operations cannot be executed, $P_0$ and $P_1$ are deadlocked.

We say that a set of processes is in a deadlocked state when *every process in the set is waiting for an event that can be caused only by another process in the set*. The "events" with which we are mainly concerned here are the acquisition and release of resources such as mutex locks and semaphores. Other types of events may result in deadlocks, as we show in more detail in Chapter 8. In that chapter, we describe various mechanisms for dealing with the deadlock problem, as well as other forms of liveness failures.

### 6.8.2 Priority Inversion

A scheduling challenge arises when a higher-priority process needs to read or modify kernel data that are currently being accessed by a lower-priority process—or a chain of lower-priority processes. Since kernel data are typically protected with a lock, the higher-priority process will have to wait for a lower-priority one to finish with the resource. The situation becomes more complicated if the lower-priority process is preempted in favor of another process with a higher priority.

As an example, assume we have three processes—$L$, $M$, and $H$—whose priorities follow the order $L < M < H$. Assume that process $H$ requires a semaphore $S$, which is currently being accessed by process $L$. Ordinarily, process $H$ would wait for $L$ to finish using resource $S$. However, now suppose that process $M$ becomes runnable, thereby preempting process $L$. Indirectly, a process with a lower priority—process $M$—has affected how long process $H$ must wait for $L$ to relinquish resource $S$.

This liveness problem is known as **priority inversion**, and it can occur only in systems with more than two priorities. Typically, priority inversion is avoided by implementing a **priority-inheritance protocol**. According to this protocol, all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question. When they are finished, their priorities revert to their original values. In the example above, a priority-inheritance protocol would allow process $L$ to temporarily inherit the priority of process $H$, thereby preventing process $M$ from preempting its execution. When process $L$ had finished using resource $S$, it would relinquish its inherited priority from $H$ and assume its original priority. Because resource $S$ would now be available, process $H$—not $M$—would run next.

## 6.9 Evaluation

We have described several different synchronization tools that can be used to solve the critical-section problem. Given correct implementation and usage, these tools can be used effectively to ensure mutual exclusion as well as address liveness issues. With the growth of concurrent programs that leverage the power of modern multicore computer systems, increasing attention is being paid to the performance of synchronization tools. Trying to identify when to use which tool, however, can be a daunting challenge. In this section, we present some simple strategies for determining when to use specific synchronization tools.

The hardware solutions outlined in Section 6.4 are considered very low level and are typically used as the foundations for constructing other synchronization tools, such as mutex locks. However, there has been a recent focus on using the CAS instruction to construct **lock-free** algorithms that provide protection from race conditions without requiring the overhead of locking. Although these lock-free solutions are gaining popularity due to low overhead

### *PRIORITY INVERSION AND THE MARS PATHFINDER*

Priority inversion can be more than a scheduling inconvenience. On systems with tight time constraints—such as real-time systems—priority inversion can cause a process to take longer than it should to accomplish a task. When that happens, other failures can cascade, resulting in system failure.

Consider the Mars Pathfinder, a NASA space probe that landed a robot, the Sojourner rover, on Mars in 1997 to conduct experiments. Shortly after the Sojourner began operating, it started to experience frequent computer resets. Each reset reinitialized all hardware and software, including communications. If the problem had not been solved, the Sojourner would have failed in its mission.

The problem was caused by the fact that one high-priority task, "bc_dist," was taking longer than expected to complete its work. This task was being forced to wait for a shared resource that was held by the lower-priority "ASI/MET" task, which in turn was preempted by multiple medium-priority tasks. The "bc_dist" task would stall waiting for the shared resource, and ultimately the "bc_sched" task would discover the problem and perform the reset. The Sojourner was suffering from a typical case of priority inversion.

The operating system on the Sojourner was the VxWorks real-time operating system, which had a global variable to enable priority inheritance on all semaphores. After testing, the variable was set on the Sojourner (on Mars!), and the problem was solved.

A full description of the problem, its detection, and its solution was written by the software team lead and is available at http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/authoritative_account.html.

and ability to scale, the algorithms themselves are often difficult to develop and test. (In the exercises at the end of this chapter, we ask you to evaluate the correctness of a lock-free stack.)

CAS-based approaches are considered an *optimistic* approach—you optimistically first update a variable and then use collision detection to see if another thread is updating the variable concurrently. If so, you repeatedly retry the operation until it is successfully updated without conflict. Mutual-exclusion locking, in contrast, is considered a *pessimistic* strategy; you assume another thread is concurrently updating the variable, so you pessimistically acquire the lock before making any updates.

The following guidelines identify general rules concerning performance differences between CAS-based synchronization and traditional synchronization (such as mutex locks and semaphores) under varying contention loads:

- **Uncontended**. Although both options are generally fast, CAS protection will be somewhat faster than traditional synchronization.
- **Moderate contention**. CAS protection will be faster—possibly much faster—than traditional synchronization.

- **High contention**. Under very highly contended loads, traditional synchronization will ultimately be faster than CAS-based synchronization.

Moderate contention is particularly interesting to examine. In this scenario, the CAS operation succeeds most of the time, and when it fails, it will iterate through the loop shown in Figure 6.8 only a few times before ultimately succeeding. By comparison, with mutual-exclusion locking, *any* attempt to acquire a contended lock will result in a more complicated—and time-intensive—code path that suspends a thread and places it on a wait queue, requiring a context switch to another thread.

The choice of a mechanism that addresses race conditions can also greatly affect system performance. For example, atomic integers are much lighter weight than traditional locks, and are generally more appropriate than mutex locks or semaphores for single updates to shared variables such as counters. We also see this in the design of operating systems where spinlocks are used on multiprocessor systems when locks are held for short durations. In general, mutex locks are simpler and require less overhead than semaphores and are preferable to binary semaphores for protecting access to a critical section. However, for some uses—such as controlling access to a finite number of resources—a counting semaphore is generally more appropriate than a mutex lock. Similarly, in some instances, a reader–writer lock may be preferred over a mutex lock, as it allows a higher degree of concurrency (that is, multiple readers).

The appeal of higher-level tools such as monitors and condition variables is based on their simplicity and ease of use. However, such tools may have significant overhead and, depending on their implementation, may be less likely to scale in highly contended situations.

Fortunately, there is much ongoing research toward developing scalable, efficient tools that address the demands of concurrent programming. Some examples include:

- Designing compilers that generate more efficient code.
- Developing languages that provide support for concurrent programming.
- Improving the performance of existing libraries and APIs.

In the next chapter, we examine how various operating systems and APIs available to developers implement the synchronization tools presented in this chapter.

## 6.10 Summary

- A race condition occurs when processes have concurrent access to shared data and the final result depends on the particular order in which concurrent accesses occur. Race conditions can result in corrupted values of shared data.
- A critical section is a section of code where shared data may be manipulated and a possible race condition may occur. The critical-section problem

is to design a protocol whereby processes can synchronize their activity to cooperatively share data.

- A solution to the critical-section problem must satisfy the following three requirements: (1) mutual exclusion, (2) progress, and (3) bounded waiting. Mutual exclusion ensures that only one process at a time is active in its critical section. Progress ensures that programs will cooperatively determine what process will next enter its critical section. Bounded waiting limits how much time a program will wait before it can enter its critical section.
- Software solutions to the critical-section problem, such as Peterson's solution, do not work well on modern computer architectures.
- Hardware support for the critical-section problem includes memory barriers; hardware instructions, such as the compare-and-swap instruction; and atomic variables.
- A mutex lock provides mutual exclusion by requiring that a process acquire a lock before entering a critical section and release the lock on exiting the critical section.
- Semaphores, like mutex locks, can be used to provide mutual exclusion. However, whereas a mutex lock has a binary value that indicates if the lock is available or not, a semaphore has an integer value and can therefore be used to solve a variety of synchronization problems.
- A monitor is an abstract data type that provides a high-level form of process synchronization. A monitor uses condition variables that allow processes to wait for certain conditions to become true and to signal one another when conditions have been set to true.
- Solutions to the critical-section problem may suffer from liveness problems, including deadlock.
- The various tools that can be used to solve the critical-section problem as well as to synchronize the activity of processes can be evaluated under varying levels of contention. Some tools work better under certain contention loads than others.

## Practice Exercises

**6.1** In Section 6.4, we mentioned that disabling interrupts frequently can affect the system's clock. Explain why this can occur and how such effects can be minimized.

**6.2** What is the meaning of the term ***busy waiting***? What other kinds of waiting are there in an operating system? Can busy waiting be avoided altogether? Explain your answer.

**6.3** Explain why spinlocks are not appropriate for single-processor systems yet are often used in multiprocessor systems.

**6.4** Show that, if the `wait()` and `signal()` semaphore operations are not executed atomically, then mutual exclusion may be violated.

**6.5** Illustrate how a binary semaphore can be used to implement mutual exclusion among *n* processes.

**6.6** Race conditions are possible in many computer systems. Consider a banking system that maintains an account balance with two functions: `deposit(amount)` and `withdraw(amount)`. These two functions are passed the `amount` that is to be deposited or withdrawn from the bank account balance. Assume that a husband and wife share a bank account. Concurrently, the husband calls the `withdraw()` function, and the wife calls `deposit()`. Describe how a race condition is possible and what might be done to prevent the race condition from occurring.

## Further Reading

The mutual-exclusion problem was first discussed in a classic paper by [Dijkstra (1965)]. The semaphore concept was suggested by [Dijkstra (1965)]. The monitor concept was developed by [Brinch-Hansen (1973)]. [Hoare (1974)] gave a complete description of the monitor.

For more on the Mars Pathfinder problem see http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/authoritative_account.html

A thorough discussion of memory barriers and cache memory is presented in [Mckenney (2010)]. [Herlihy and Shavit (2012)] presents details on several issues related to multiprocessor programming, including memory models and compare-and-swap instructions. [Bahra (2013)] examines nonblocking algorithms on modern multicore systems.

## Bibliography

**[Bahra (2013)]** S. A. Bahra, “Nonblocking Algorithms and Scalable Multicore Programming”, *ACM queue*, Volume 11, Number 5 (2013).

**[Brinch-Hansen (1973)]** P. Brinch-Hansen, *Operating System Principles*, Prentice Hall (1973).

**[Dijkstra (1965)]** E. W. Dijkstra, “Cooperating Sequential Processes”, Technical report, Technological University, Eindhoven, the Netherlands (1965).

**[Herlihy and Shavit (2012)]** M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming,* Revised First Edition, Morgan Kaufmann Publishers Inc. (2012).

**[Hoare (1974)]** C. A. R. Hoare, “Monitors: An Operating System Structuring Concept”, *Communications of the ACM*, Volume 17, Number 10 (1974), pages 549–557.

**[Mckenney (2010)]** P. E. Mckenney, “Memory Barriers: a Hardware View for Software Hackers” (2010).

# Chapter 6 Exercises

**6.7** The pseudocode of Figure 6.15 illustrates the basic `push()` and `pop()` operations of an array-based stack. Assuming that this algorithm could be used in a concurrent environment, answer the following questions:

a. What data have a race condition?

b. How could the race condition be fixed?

**6.8** Race conditions are possible in many computer systems. Consider an online auction system where the current highest bid for each item must be maintained. A person who wishes to bid on an item calls the `bid(amount)` function, which compares the amount being bid to the current highest bid. If the amount exceeds the current highest bid, the highest bid is set to the new amount. This is illustrated below:

```
void bid(double amount) {
  if (amount > highestBid)
    highestBid = amount;
}
```

```
push(item) {
  if (top < SIZE) {
    stack[top] = item;
    top++;
  }
  else
    ERROR
}

pop() {
  if (!is_empty()) {
    top--;
    return stack[top];
  }
  else
    ERROR
}

is_empty() {
  if (top == 0)
    return true;
  else
    return false;
}
```

**Figure 6.16** Array-based stack for Exercise 6.12.

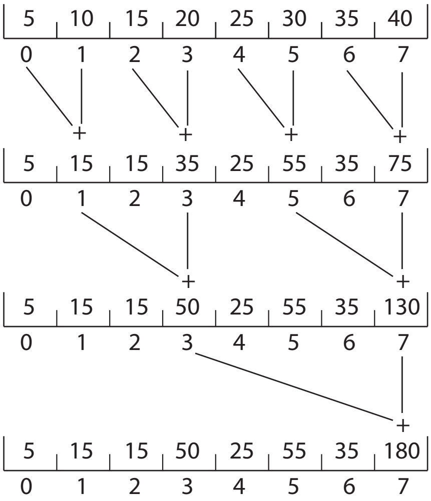


**Figure 6.17** Summing an array as a series of partial sums for Exercise 6.14.

Describe how a race condition is possible in this situation and what might be done to prevent the race condition from occurring.

**6.9** The following program example can be used to sum the array `values` of size *N* elements in parallel on a system containing *N* computing cores (there is a separate processor for each array element):

```
for j = 1 to log_2(N) {
   for k = 1 to N {
      if ((k + 1) % pow(2,j) == 0) {
         values[k] += values[k - pow(2,(j-1))]
      }
   }
}
```

This has the effect of summing the elements in the array as a series of partial sums, as shown in Figure 6.16. After the code has executed, the sum of all elements in the array is stored in the last array location. Are there any race conditions in the above code example? If so, identify where they occur and illustrate with an example. If not, demonstrate why this algorithm is free from race conditions.

**6.10** The `compare_and_swap()` instruction can be used to design lock-free data structures such as stacks, queues, and lists. The program example shown in Figure 6.17 presents a possible solution to a lock-free stack using CAS instructions, where the stack is represented as a linked list of `Node` elements with `top` representing the top of the stack. Is this implementation free from race conditions?

```
typedef struct node {
   value_t data;
   struct node *next;
} Node;

Node *top; // top of stack

void push(value_t item) {
   Node *old_node;
   Node *new_node;

   new_node = malloc(sizeof(Node));
   new_node->data = item;

   do {
      old_node = top;
      new_node->next = old_node;
   }
   while (compare_and_swap(top,old_node,new_node) != old_node);
}

value_t pop() {
   Node *old_node;
   Node *new_node;

   do {
      old_node = top;
      if (old_node == NULL)
         return NULL;
      new_node = old_node->next;
   }
   while (compare_and_swap(top,old_node,new_node) != old_node);

   return old_node->data;
}
```

**Figure 6.18** Lock-free stack for Exercise 6.15.

**6.11** One approach for using `compare_and_swap()` for implementing a spinlock is as follows:

```
void lock_spinlock(int *lock) {
   while (compare_and_swap(lock, 0, 1) != 0)
      ; /* spin */
}
```

A suggested alternative approach is to use the "compare and compare-and-swap" idiom, which checks the status of the lock before invoking the `compare_and_swap()` operation. (The rationale behind this approach is to invoke `compare_and_swap()`only if the lock is currently available.) This strategy is shown below:

```
void lock_spinlock(int *lock) {
{
   while (true) {
      if (*lock == 0) {
         /* lock appears to be available */

         if (!compare_and_swap(lock, 0, 1))
            break;
      }
   }
}
```

Does this "compare and compare-and-swap" idiom work appropriately for implementing spinlocks? If so, explain. If not, illustrate how the integrity of the lock is compromised.

**6.12** Some semaphore implementations provide a function `getValue()` that returns the current value of a semaphore. This function may, for instance, be invoked prior to calling `wait()` so that a process will only call `wait()` if the value of the semaphore is > 0, thereby preventing blocking while waiting for the semaphore. For example:

```
if (getValue(&sem) > 0)
   wait(&sem);
```

Many developers argue against such a function and discourage its use. Describe a potential problem that could occur when using the function `getValue()` in this scenario.

**6.13** The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, $P_0$ and $P_1$, share the following variables:

```
boolean flag[2]; /* initially false */
int turn;
```

The structure of process $P_i$ (`i == 0` or 1) is shown in Figure 6.18. The other process is $P_j$ (`j == 1` or 0). Prove that the algorithm satisfies all three requirements for the critical-section problem.

**6.14** The first known correct software solution to the critical-section problem for $n$ processes with a lower bound on waiting of $n - 1$ turns was presented by Eisenberg and McGuire. The processes share the following variables:

```
enum pstate {idle, want_in, in_cs};
pstate flag[n];
int turn;
```

```
while (true) {
   flag[i] = true;

   while (flag[j]) {
      if (turn == j) {
         flag[i] = false;
         while (turn == j)
            ; /* do nothing */
         flag[i] = true;
      }
   }

      /* critical section */

   turn = j;
   flag[i] = false;

      /* remainder section */
}
```

**Figure 6.19** The structure of process $P_i$ in Dekker's algorithm.

All the elements of `flag` are initially `idle`. The initial value of `turn` is immaterial (between 0 and `n-1`). The structure of process $P_i$ is shown in Figure 6.19. Prove that the algorithm satisfies all three requirements for the critical-section problem.

**6.15** Explain why implementing synchronization primitives by disabling interrupts is not appropriate in a single-processor system if the synchronization primitives are to be used in user-level programs.

**6.16** Consider how to implement a mutex lock using the `compare_and_swap()` instruction. Assume that the following structure defining the mutex lock is available:

```
typedef struct {
      int available;
} lock;
```

The value (`available == 0`) indicates that the lock is available, and a value of 1 indicates that the lock is unavailable. Using this `struct`, illustrate how the following functions can be implemented using the `compare_and_swap()` instruction:

- `void acquire(lock *mutex)`
- `void release(lock *mutex)`

Be sure to include any initialization that may be necessary.

```
while (true) {
  while (true) {
    flag[i] = want_in;
    j = turn;

    while (j != i) {
      if (flag[j] != idle) {
        j = turn;
      else
        j = (j + 1) % n;
    }

    flag[i] = in_cs;
    j = 0;

    while ( (j < n) && (j == i || flag[j] != in_cs))
      j++;

    if ( (j >= n) && (turn == i || flag[turn] == idle))
      break;
  }

    /* critical section */

  j = (turn + 1) % n;

  while (flag[j] == idle)
    j = (j + 1) % n;

  turn = j;
  flag[i] = idle;

    /* remainder section */
}
```

**Figure 6.20** The structure of process $P_i$ in Eisenberg and McGuire's algorithm.

**6.17** Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems.

**6.18** The implementation of mutex locks provided in Section 6.5 suffers from busy waiting. Describe what changes would be necessary so that a process waiting to acquire a mutex lock would be blocked and placed into a waiting queue until the lock became available.

**6.19** Assume that a system has multiple processing cores. For each of the following scenarios, describe which is a better locking mechanism—a spinlock or a mutex lock where waiting processes sleep while waiting for the lock to become available:

- The lock is to be held for a short duration.
- The lock is to be held for a long duration.
- A thread may be put to sleep while holding the lock.

**6.20** Assume that a context switch takes *T* time. Suggest an upper bound (in terms of *T*) for holding a spinlock. If the spinlock is held for any longer, a mutex lock (where waiting threads are put to sleep) is a better alternative.

**6.21** A multithreaded web server wishes to keep track of the number of requests it services (known as ***hits***). Consider the two following strategies to prevent a race condition on the variable `hits`. The first strategy is to use a basic mutex lock when updating `hits`:

```
int hits;
mutex_lock hit_lock;

hit_lock.acquire();
hits++;
hit_lock.release();
```

A second strategy is to use an atomic integer:

```
atomic_t hits;
atomic_inc(&hits);
```

Explain which of these two strategies is more efficient.

**6.22** Consider the code example for allocating and releasing processes shown in Figure 6.20.

a. Identify the race condition(s).

b. Assume you have a mutex lock named `mutex` with the operations `acquire()` and `release()`. Indicate where the locking needs to be placed to prevent the race condition(s).

c. Could we replace the integer variable

```
int number_of_processes = 0
```

with the atomic integer

```
atomic_t number_of_processes = 0
```

to prevent the race condition(s)?

**6.23** Servers can be designed to limit the number of open connections. For example, a server may wish to have only *N* socket connections at any point in time. As soon as *N* connections are made, the server will not accept another incoming connection until an existing connection is

```
#define MAX_PROCESSES 255
int number_of_processes = 0;

/* the implementation of fork() calls this function */
int allocate_process() {
int new_pid;

   if (number_of_processes == MAX_PROCESSES)
      return -1;
   else {
      /* allocate necessary process resources */
      ++number_of_processes;

      return new_pid;
   }
}

/* the implementation of exit() calls this function */
void release_process() {
    /* release process resources */
    --number_of_processes;
}
```

**Figure 6.21** Allocating and releasing processes for Exercise 6.27.

released. Illustrate how semaphores can be used by a server to limit the number of concurrent connections.

**6.24** In Section 6.7, we use the following illustration as an incorrect use of semaphores to solve the critical-section problem:

```
wait(mutex);
     ...
  critical section
     ...
wait(mutex);
```

Explain why this is an example of a liveness failure.

**6.25** Demonstrate that monitors and semaphores are equivalent to the degree that they can be used to implement solutions to the same types of synchronization problems.

**6.26** Describe how the `signal()` operation associated with monitors differs from the corresponding operation defined for semaphores.

**6.27** Suppose the `signal()` statement can appear only as the last statement in a monitor function. Suggest how the implementation described in Section 6.7 can be simplified in this situation.

**6.28** Consider a system consisting of processes $P_1, P_2, ..., P_n$, each of which has a unique priority number. Write a monitor that allocates three identical printers to these processes, using the priority numbers for deciding the order of allocation.

**6.29** A file is to be shared among different processes, each of which has a unique number. The file can be accessed simultaneously by several processes, subject to the following constraint: the sum of all unique numbers associated with all the processes currently accessing the file must be less than $n$. Write a monitor to coordinate access to the file.

**6.30** When a signal is performed on a condition inside a monitor, the signaling process can either continue its execution or transfer control to the process that is signaled. How would the solution to the preceding exercise differ with these two different ways in which signaling can be performed?

**6.31** Design an algorithm for a monitor that implements an alarm clock that enables a calling program to delay itself for a specified number of time units (*ticks*). You may assume the existence of a real hardware clock that invokes a function `tick()` in your monitor at regular intervals.

**6.32** Discuss ways in which the priority inversion problem could be addressed in a real-time system. Also discuss whether the solutions could be implemented within the context of a proportional share scheduler.

## Programming Problems

**6.33** Assume that a finite number of resources of a single resource type must be managed. Processes may ask for a number of these resources and will return them once finished. As an example, many commercial software packages provide a given number of ***licenses***, indicating the number of applications that may run concurrently. When the application is started, the license count is decremented. When the application is terminated, the license count is incremented. If all licenses are in use, requests to start the application are denied. Such a request will be granted only when an existing license holder terminates the application and a license is returned.

The following program segment is used to manage a finite number of instances of an available resource. The maximum number of resources and the number of available resources are declared as follows:

```
#define MAX_RESOURCES 5
int available_resources = MAX_RESOURCES;
```

When a process wishes to obtain a number of resources, it invokes the `decrease_count()` function:

```
/* decrease available_resources by count resources */
/* return 0 if sufficient resources available, */
/* otherwise return -1 */
int decrease_count(int count) {
  if (available_resources < count)
    return -1;
  else {
    available_resources -= count;

    return 0;
  }
}
```

When a process wants to return a number of resources, it calls the `increase_count()` function:

```
/* increase available_resources by count */
int increase_count(int count) {
  available_resources += count;

  return 0;
}
```

The preceding program segment produces a race condition. Do the following:

a. Identify the data involved in the race condition.

b. Identify the location (or locations) in the code where the race condition occurs.

c. Using a semaphore or mutex lock, fix the race condition. It is permissible to modify the `decrease_count()` function so that the calling process is blocked until sufficient resources are available.

**6.34** The `decrease_count()` function in the previous exercise currently returns 0 if sufficient resources are available and −1 otherwise. This leads to awkward programming for a process that wishes to obtain a number of resources:

```
while (decrease_count(count) == -1)
  ;
```

Rewrite the resource-manager code segment using a monitor and condition variables so that the `decrease_count()` function suspends the process until sufficient resources are available. This will allow a process to invoke `decrease_count()` by simply calling

```
decrease_count(count);
```

The process will return from this function call only when sufficient resources are available.

# CHAPTER 7

# Synchronization Examples

In Chapter 6, we presented the critical-section problem and focused on how race conditions can occur when multiple concurrent processes share data. We went on to examine several tools that address the critical-section problem by preventing race conditions from occurring. These tools ranged from low-level hardware solutions (such as memory barriers and the compare-and-swap operation) to increasingly higher-level tools (from mutex locks to semaphores to monitors). We also discussed various challenges in designing applications that are free from race conditions, including liveness hazards such as deadlocks. In this chapter, we apply the tools presented in Chapter 6 to several classic synchronization problems. We also explore the synchronization mechanisms used by the Linux, UNIX, and Windows operating systems, and we describe API details for both Java and POSIX systems.

## CHAPTER OBJECTIVES

- Explain the bounded-buffer, readers–writers, and dining–philosophers synchronization problems.
- Describe specific tools used by Linux and Windows to solve process synchronization problems.
- Illustrate how POSIX and Java can be used to solve process synchronization problems.
- Design and develop solutions to process synchronization problems using POSIX and Java APIs.

## 7.1 Classic Problems of Synchronization

In this section, we present a number of synchronization problems as examples of a large class of concurrency-control problems. These problems are used for testing nearly every newly proposed synchronization scheme. In our solutions to the problems, we use semaphores for synchronization, since that is the

```
while (true) {
      . . .
   /* produce an item in next_produced */
      . . .
   wait(empty);
   wait(mutex);
      . . .
   /* add next_produced to the buffer */
      . . .
   signal(mutex);
   signal(full);
}
```

**Figure 7.1** The structure of the producer process.

traditional way to present such solutions. However, actual implementations of these solutions could use mutex locks in place of binary semaphores.

### 7.1.1 The Bounded-Buffer Problem

The ***bounded-buffer problem*** was introduced in Section 6.1; it is commonly used to illustrate the power of synchronization primitives. Here, we present a general structure of this scheme without committing ourselves to any particular implementation. We provide a related programming project in the exercises at the end of the chapter.

In our problem, the producer and consumer processes share the following data structures:

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

We assume that the pool consists of `n` buffers, each capable of holding one item. The `mutex` binary semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The `empty` and `full` semaphores count the number of empty and full buffers. The semaphore `empty` is initialized to the value `n`; the semaphore `full` is initialized to the value 0.

The code for the producer process is shown in Figure 7.1, and the code for the consumer process is shown in Figure 7.2. Note the symmetry between the producer and the consumer. We can interpret this code as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

### 7.1.2 The Readers–Writers Problem

Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, read and write) the database. We distinguish

```
while (true) {
   wait(full);
   wait(mutex);
      . . .
   /* remove an item from buffer to next_consumed */
      . . .
   signal(mutex);
   signal(empty);
      . . .
   /* consume the item in next_consumed */
      . . .
}
```

**Figure 7.2** The structure of the consumer process.

between these two types of processes by referring to the former as ***readers*** and to the latter as ***writers***. Obviously, if two readers access the shared data simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may ensue.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database. This synchronization problem is referred to as the **readers–writers problem**. Since it was originally stated, it has been used to test nearly every new synchronization primitive.

The readers–writers problem has several variations, all involving priorities. The simplest one, referred to as the ***first*** readers–writers problem, requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting. The ***second*** readers–writers problem requires that, once a writer is ready, that writer perform its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve. For this reason, other variants of the problem have been proposed. Next, we present a solution to the first readers–writers problem. See the bibliographical notes at the end of the chapter for references describing starvation-free solutions to the second readers–writers problem.

In the solution to the first readers–writers problem, the reader processes share the following data structures:

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
```

The binary semaphores `mutex` and `rw_mutex` are initialized to 1; `read_count` is a counting semaphore initialized to 0. The semaphore `rw_mutex`

```
while (true) {
   wait(rw_mutex);
      . . .
   /* writing is performed */
      . . .
   signal(rw_mutex);
}
```

**Figure 7.3** The structure of a writer process.

is common to both reader and writer processes. The `mutex` semaphore is used to ensure mutual exclusion when the variable `read_count` is updated. The `read_count` variable keeps track of how many processes are currently reading the object. The semaphore `rw_mutex` functions as a mutual exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section. It is not used by readers that enter or exit while other readers are in their critical sections.

The code for a writer process is shown in Figure 7.3; the code for a reader process is shown in Figure 7.4. Note that, if a writer is in the critical section and $n$ readers are waiting, then one reader is queued on `rw_mutex`, and $n - 1$ readers are queued on `mutex`. Also observe that, when a writer executes `signal(rw_mutex)`, we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler.

The readers–writers problem and its solutions have been generalized to provide **reader–writer** locks on some systems. Acquiring a reader–writer lock requires specifying the mode of the lock: either *read* or *write* access. When a

```
while (true) {
   wait(mutex);
   read_count++;
   if (read_count == 1)
      wait(rw_mutex);
   signal(mutex);
      . . .
   /* reading is performed */
      . . .
   wait(mutex);
   read_count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
}
```

**Figure 7.4** The structure of a reader process.

process wishes only to read shared data, it requests the reader–writer lock in read mode. A process wishing to modify the shared data must request the lock in write mode. Multiple processes are permitted to concurrently acquire a reader–writer lock in read mode, but only one process may acquire the lock for writing, as exclusive access is required for writers.

Reader–writer locks are most useful in the following situations:

- In applications where it is easy to identify which processes only read shared data and which processes only write shared data.
- In applications that have more readers than writers. This is because reader–writer locks generally require more overhead to establish than semaphores or mutual-exclusion locks. The increased concurrency of allowing multiple readers compensates for the overhead involved in setting up the reader–writer lock.

### 7.1.3 The Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks (Figure 7.5). When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking again.

The **dining-philosophers problem** is considered a classic synchronization problem neither because of its practical importance nor because computer scientists dislike philosophers but because it is an example of a large class of concurrency-control problems. It is a simple representation of the need

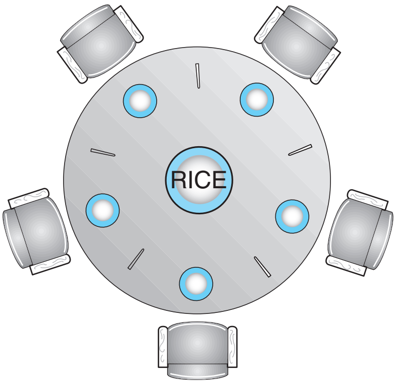


**Figure 7.5** The situation of the dining philosophers.

```
while (true) {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for a while */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
}
```

**Figure 7.6** The structure of philosopher *i*.

to allocate several resources among several processes in a deadlock-free and starvation-free manner.

#### 7.1.3.1 Semaphore Solution

One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a `wait()` operation on that semaphore. She releases her chopsticks by executing the `signal()` operation on the appropriate semaphores. Thus, the shared data are

```
semaphore chopstick[5];
```

where all the elements of `chopstick` are initialized to 1. The structure of philosopher *i* is shown in Figure 7.6.

Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock. Suppose that all five philosophers become hungry at the same time and each grabs her left chopstick. All the elements of `chopstick` will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

Several possible remedies to the deadlock problem are the following:

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
- Use an asymmetric solution—that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even-numbered philosopher picks up her right chopstick and then her left chopstick.

In Section 6.7, we present a solution to the dining-philosophers problem that ensures freedom from deadlocks. Note, however, that any satisfactory solution to the dining-philosophers problem must guard against the possibility that one of the philosophers will starve to death. A deadlock-free solution does not necessarily eliminate the possibility of starvation.

#### 7.1.3.2 Monitor Solution

Next, we illustrate monitor concepts by presenting a deadlock-free solution to the dining-philosophers problem. This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available. To code this solution, we need to distinguish among three states in which we may find a philosopher. For this purpose, we introduce the following data structure:

```
enum {THINKING, HUNGRY, EATING} state[5];
```

Philosopher *i* can set the variable `state[i] = EATING` only if her two neighbors are not eating: (`state[(i+4) % 5] != EATING`) and (`state[(i+1) % 5] != EATING`).

We also need to declare

```
condition self[5];
```

This allows philosopher *i* to delay herself when she is hungry but is unable to obtain the chopsticks she needs.

We are now in a position to describe our solution to the dining-philosophers problem. The distribution of the chopsticks is controlled by the monitor `DiningPhilosophers`, whose definition is shown in Figure 7.7. Each philosopher, before starting to eat, must invoke the operation `pickup()`. This act may result in the suspension of the philosopher process. After the successful completion of the operation, the philosopher may eat. Following this, the philosopher invokes the `putdown()` operation. Thus, philosopher *i* must invoke the operations `pickup()` and `putdown()` in the following sequence:

```
DiningPhilosophers.pickup(i);
              ...
              eat
              ...
DiningPhilosophers.putdown(i);
```

It is easy to show that this solution ensures that no two neighbors are eating simultaneously and that no deadlocks will occur. As we already noted, however, it is possible for a philosopher to starve to death. We do not present a solution to this problem but rather leave it as an exercise for you.

## 7.2 Synchronization within the Kernel

We next describe the synchronization mechanisms provided by the Windows and Linux operating systems. These two operating systems provide good examples of different approaches to synchronizing the kernel, and as you will

```
monitor DiningPhilosophers
{
   enum {THINKING, HUNGRY, EATING} state[5];
   condition self[5];

   void pickup(int i) {
      state[i] = HUNGRY;
      test(i);
      if (state[i] != EATING)
         self[i].wait();
   }

   void putdown(int i) {
      state[i] = THINKING;
      test((i + 4) % 5);
      test((i + 1) % 5);
   }

   void test(int i) {
      if ((state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING)) {
           state[i] = EATING;
           self[i].signal();
      }
   }

   initialization_code() {
      for (int i = 0; i < 5; i++)
         state[i] = THINKING;
    }
}
```

**Figure 7.7** A monitor solution to the dining-philosophers problem.

see, the synchronization mechanisms available in these systems differ in subtle yet significant ways.

### 7.2.1 Synchronization in Windows

The Windows operating system is a multithreaded kernel that provides support for real-time applications and multiple processors. When the Windows kernel accesses a global resource on a single-processor system, it temporarily masks interrupts for all interrupt handlers that may also access the global resource. On a multiprocessor system, Windows protects access to global resources using spinlocks, although the kernel uses spinlocks only to protect short code segments. Furthermore, for reasons of efficiency, the kernel ensures that a thread will never be preempted while holding a spinlock.

For thread synchronization outside the kernel, Windows provides **dispatcher objects**. Using a dispatcher object, threads synchronize according to several different mechanisms, including mutex locks, semaphores, events, and timers. The system protects shared data by requiring a thread to gain ownership of a mutex to access the data and to release ownership when it is finished. Semaphores behave as described in Section 6.6. **Events** are similar to condition variables; that is, they may notify a waiting thread when a desired condition occurs. Finally, timers are used to notify one (or more than one) thread that a specified amount of time has expired.

Dispatcher objects may be in either a signaled state or a nonsignaled state. An object in a **signaled state** is available, and a thread will not block when acquiring the object. An object in a **nonsignaled state** is not available, and a thread will block when attempting to acquire the object. We illustrate the state transitions of a mutex lock dispatcher object in Figure 7.8.

A relationship exists between the state of a dispatcher object and the state of a thread. When a thread blocks on a nonsignaled dispatcher object, its state changes from ready to waiting, and the thread is placed in a waiting queue for that object. When the state for the dispatcher object moves to signaled, the kernel checks whether any threads are waiting on the object. If so, the kernel moves one thread—or possibly more—from the waiting state to the ready state, where they can resume executing. The number of threads the kernel selects from the waiting queue depends on the type of dispatcher object for which each thread is waiting. The kernel will select only one thread from the waiting queue for a mutex, since a mutex object may be "owned" by only a single thread. For an event object, the kernel will select all threads that are waiting for the event.

We can use a mutex lock as an illustration of dispatcher objects and thread states. If a thread tries to acquire a mutex dispatcher object that is in a nonsignaled state, that thread will be suspended and placed in a waiting queue for the mutex object. When the mutex moves to the signaled state (because another thread has released the lock on the mutex), the thread waiting at the front of the queue will be moved from the waiting state to the ready state and will acquire the mutex lock.

A **critical-section object** is a user-mode mutex that can often be acquired and released without kernel intervention. On a multiprocessor system, a critical-section object first uses a spinlock while waiting for the other thread to release the object. If it spins too long, the acquiring thread will then allocate a kernel mutex and yield its CPU. Critical-section objects are particularly efficient because the kernel mutex is allocated only when there is contention for the object. In practice, there is very little contention, so the savings are significant.

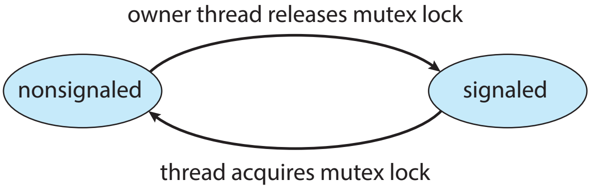


**Figure 7.8** Mutex dispatcher object.

We provide a programming project at the end of this chapter that uses mutex locks and semaphores in the Windows API.

### 7.2.2 Synchronization in Linux

Prior to Version 2.6, Linux was a nonpreemptive kernel, meaning that a process running in kernel mode could not be preempted—even if a higher-priority process became available to run. Now, however, the Linux kernel is fully preemptive, so a task can be preempted when it is running in the kernel.

Linux provides several different mechanisms for synchronization in the kernel. As most computer architectures provide instructions for atomic versions of simple math operations, the simplest synchronization technique within the Linux kernel is an atomic integer, which is represented using the opaque data type `atomic_t`. As the name implies, all math operations using atomic integers are performed without interruption. To illustrate, consider a program that consists of an atomic integer `counter` and an integer `value`.

```
atomic_t counter;
int value;
```

The following code illustrates the effect of performing various atomic operations:

| *Atomic Operation* | *Effect* |
|---|---|
| `atomic_set(&counter,5);` | `counter = 5` |
| `atomic_add(10,&counter);` | `counter = counter + 10` |
| `atomic_sub(4,&counter);` | `counter = counter - 4` |
| `atomic_inc(&counter);` | `counter = counter + 1` |
| `value = atomic_read(&counter);` | `value = 12` |

Atomic integers are particularly efficient in situations where an integer variable—such as a counter—needs to be updated, since atomic operations do not require the overhead of locking mechanisms. However, their use is limited to these sorts of scenarios. In situations where there are several variables contributing to a possible race condition, more sophisticated locking tools must be used.

Mutex locks are available in Linux for protecting critical sections within the kernel. Here, a task must invoke the `mutex_lock()` function prior to entering a critical section and the `mutex_unlock()` function after exiting the critical section. If the mutex lock is unavailable, a task calling `mutex_lock()` is put into a sleep state and is awakened when the lock's owner invokes `mutex_unlock()`.

Linux also provides spinlocks and semaphores (as well as reader–writer versions of these two locks) for locking in the kernel. On SMP machines, the fundamental locking mechanism is a spinlock, and the kernel is designed so that the spinlock is held only for short durations. On single-processor machines, such as embedded systems with only a single processing core, spinlocks are inappropriate for use and are replaced by enabling and disabling kernel preemption. That is, on systems with a single processing core, rather than holding a spinlock, the kernel disables kernel preemption; and rather than releasing the spinlock, it enables kernel preemption. This is summarized below:

| Single Processor | Multiple Processors |
|---|---|
| Disable kernel preemption | Acquire spin lock |
| Enable kernel preemption | Release spin lock |

In the Linux kernel, both spinlocks and mutex locks are ***nonrecursive***, which means that if a thread has acquired one of these locks, it cannot acquire the same lock a second time without first releasing the lock. Otherwise, the second attempt at acquiring the lock will block.

Linux uses an interesting approach to disable and enable kernel preemption. It provides two simple system calls—`preempt_disable()` and `preempt_enable()`—for disabling and enabling kernel preemption. The kernel is not preemptible, however, if a task running in the kernel is holding a lock. To enforce this rule, each task in the system has a `thread-info` structure containing a counter, `preempt_count`, to indicate the number of locks being held by the task. When a lock is acquired, `preempt_count` is incremented. It is decremented when a lock is released. If the value of `preempt_count` for the task currently running in the kernel is greater than 0, it is not safe to preempt the kernel, as this task currently holds a lock. If the count is 0, the kernel can safely be interrupted (assuming there are no outstanding calls to `preempt_disable()`).

Spinlocks—along with enabling and disabling kernel preemption—are used in the kernel only when a lock (or disabling kernel preemption) is held for a short duration. When a lock must be held for a longer period, semaphores or mutex locks are appropriate for use.

## 7.3 POSIX Synchronization

The synchronization methods discussed in the preceding section pertain to synchronization within the kernel and are therefore available only to kernel developers. In contrast, the POSIX API is available for programmers at the user level and is not part of any particular operating-system kernel. (Of course, it must ultimately be implemented using tools provided by the host operating system.)

In this section, we cover mutex locks, semaphores, and condition variables that are available in the Pthreads and POSIX APIs. These APIs are widely used for thread creation and synchronization by developers on UNIX, Linux, and macOS systems.

### 7.3.1 POSIX Mutex Locks

Mutex locks represent the fundamental synchronization technique used with Pthreads. A mutex lock is used to protect critical sections of code—that is, a thread acquires the lock before entering a critical section and releases it upon exiting the critical section. Pthreads uses the `pthread_mutex_t` data type for mutex locks. A mutex is created with the `pthread_mutex_init()` function. The first parameter is a pointer to the mutex. By passing `NULL` as a second parameter, we initialize the mutex to its default attributes. This is illustrated below:

```
#include <pthread.h>

pthread_mutex_t mutex;

/* create and initialize the mutex lock */
pthread_mutex_init(&mutex,NULL);
```

The mutex is acquired and released with the `pthread_mutex_lock()` and `pthread_mutex_unlock()` functions. If the mutex lock is unavailable when `pthread_mutex_lock()` is invoked, the calling thread is blocked until the owner invokes `pthread_mutex_unlock()`. The following code illustrates protecting a critical section with mutex locks:

```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

All mutex functions return a value of 0 with correct operation; if an error occurs, these functions return a nonzero error code.

### 7.3.2 POSIX Semaphores

Many systems that implement Pthreads also provide semaphores, although semaphores are not part of the POSIX standard and instead belong to the POSIX SEM extension. POSIX specifies two types of semaphores—**named** and **unnamed**. Fundamentally, the two are quite similar, but they differ in terms of how they are created and shared between processes. Because both techniques are common, we discuss both here. Beginning with Version 2.6 of the kernel, Linux systems provide support for both named and unnamed semaphores.

#### 7.3.2.1 POSIX Named Semaphores

The function `sem_open()` is used to create and open a POSIX named sempahore:

```
#include <semaphore.h>
sem_t *sem;

/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);
```

In this instance, we are naming the semaphore `SEM`. The `O_CREAT` flag indicates that the semaphore will be created if it does not already exist. Additionally, the semaphore has read and write access for other processes (via the parameter `0666`) and is initialized to 1.

The advantage of named semaphores is that multiple unrelated processes can easily use a common semaphore as a synchronization mechanism by simply referring to the semaphore's name. In the example above, once the semaphore `SEM` has been created, subsequent calls to `sem_open()` (with the same parameters) by other processes return a descriptor to the existing semaphore.

In Section 6.6, we described the classic `wait()` and `signal()` semaphore operations. POSIX declares these operations `sem_wait()` and `sem_post()`, respectively. The following code sample illustrates protecting a critical section using the named semaphore created above:

```
/* acquire the semaphore */
sem_wait(sem);

/* critical section */

/* release the semaphore */
sem_post(sem);
```

Both Linux and macOS systems provide POSIX named semaphores.

#### 7.3.2.2 POSIX Unnamed Semaphores

An unnamed semaphore is created and initialized using the `sem_init()` function, which is passed three parameters:

1. A pointer to the semaphore
2. A flag indicating the level of sharing
3. The semaphore's initial value

and is illusrated in the following programming example:

```
#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);
```

In this example, by passing the flag 0, we are indicating that this semaphore can be shared only by threads belonging to the process that created the semaphore. (If we supplied a nonzero value, we could allow the semaphore to be shared between separate processes by placing it in a region of shared memory.) In addition, we initialize the semaphore to the value 1.

POSIX unnamed semaphores use the same `sem_wait()` and `sem_post()` operations as named semaphores. The following code sample illustrates protecting a critical section using the unnamed semaphore created above:

```
/* acquire the semaphore */
sem_wait(&sem);

/* critical section */

/* release the semaphore */
sem_post(&sem);
```

Just like mutex locks, all semaphore functions return 0 when successful and nonzero when an error condition occurs.

### 7.3.3 POSIX Condition Variables

Condition variables in Pthreads behave similarly to those described in Section 6.7. However, in that section, condition variables are used within the context of a monitor, which provides a locking mechanism to ensure data integrity. Since Pthreads is typically used in C programs—and since C does not have a monitor— we accomplish locking by associating a condition variable with a mutex lock.

Condition variables in Pthreads use the `pthread_cond_t` data type and are initialized using the `pthread_cond_init()` function. The following code creates and initializes a condition variable as well as its associated mutex lock:

```
pthread_mutex_t mutex;
pthread_cond_t cond_var;

pthread_mutex_init(&mutex,NULL);
pthread_cond_init(&cond_var,NULL);
```

The `pthread_cond_wait()` function is used for waiting on a condition variable. The following code illustrates how a thread can wait for the condition `a == b` to become true using a Pthread condition variable:

```
pthread_mutex_lock(&mutex);
while (a != b)
     pthread_cond_wait(&cond_var, &mutex);

pthread_mutex_unlock(&mutex);
```

The mutex lock associated with the condition variable must be locked before the `pthread_cond_wait()` function is called, since it is used to protect the data in the conditional clause from a possible race condition. Once this lock is acquired, the thread can check the condition. If the condition is not true, the thread then invokes `pthread_cond_wait()`, passing the mutex lock and the condition variable as parameters. Calling `pthread_cond_wait()` releases the mutex lock, thereby allowing another thread to access the shared data and possibly update its value so that the condition clause evaluates to true. (To protect against program errors, it is important to place the conditional clause within a loop so that the condition is rechecked after being signaled.)

A thread that modifies the shared data can invoke the `pthread_cond_signal()` function, thereby signaling one thread waiting on the condition variable. This is illustrated below:

```
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```

It is important to note that the call to `pthread_cond_signal()` does not release the mutex lock. It is the subsequent call to `pthread_mutex_unlock()` that releases the mutex. Once the mutex lock is released, the signaled thread becomes the owner of the mutex lock and returns control from the call to `pthread_cond_wait()`.

We provide several programming problems and projects at the end of this chapter that use Pthreads mutex locks and condition variables, as well as POSIX semaphores.

## 7.4 Synchronization in Java

The Java language and its API have provided rich support for thread synchronization since the origins of the language. In this section, we first cover Java monitors, Java's original synchronization mechanism. We then cover three additional mechanisms that were introduced in Release 1.5: reentrant locks, semaphores, and condition variables. We include these because they represent the most common locking and synchronization mechanisms. However, the Java API provides many features that we do not cover in this text—for example, support for atomic variables and the CAS instruction—and we encourage interested readers to consult the bibliography for more information.

### 7.4.1 Java Monitors

Java provides a monitor-like concurrency mechanism for thread synchronization. We illustrate this mechanism with the `BoundedBuffer` class (Figure 7.9), which implements a solution to the bounded-buffer problem wherein the producer and consumer invoke the `insert()` and `remove()` methods, respectively.

Every object in Java has associated with it a single lock. When a method is declared to be `synchronized`, calling the method requires owning the lock for the object. We declare a `synchronized` method by placing the `synchronized` keyword in the method definition, such as with the `insert()` and `remove()` methods in the `BoundedBuffer` class.

Invoking a `synchronized` method requires owning the lock on an object instance of `BoundedBuffer`. If the lock is already owned by another thread, the thread calling the `synchronized` method blocks and is placed in the **entry set** for the object's lock. The entry set represents the set of threads waiting for the lock to become available. If the lock is available when a `synchronized` method is called, the calling thread becomes the owner of the object's lock and can enter the method. The lock is released when the thread exits the method. If the entry set for the lock is not empty when the lock is released, the JVM

```
public class BoundedBuffer<E>
{
   private static final int BUFFER_SIZE = 5;

   private int count, in, out;
   private E[] buffer;

   public BoundedBuffer() {
      count = 0;
      in = 0;
      out = 0;
      buffer = (E[]) new Object[BUFFER_SIZE];
   }

   /* Producers call this method */
   public synchronized void insert(E item) {
      /* See Figure 7.11 */
   }

   /* Consumers call this method */
   public synchronized E remove() {
      /* See Figure 7.11 */
   }
}
```

**Figure 7.9** Bounded buffer using Java synchronization.

arbitrarily selects a thread from this set to be the owner of the lock. (When we say "arbitrarily," we mean that the specification does not require that threads in this set be organized in any particular order. However, in practice, most virtual machines order threads in the entry set according to a FIFO policy.) Figure 7.10 illustrates how the entry set operates.

In addition to having a lock, every object also has associated with it a **wait set** consisting of a set of threads. This wait set is initially empty. When a thread enters a `synchronized` method, it owns the lock for the object. However, this thread may determine that it is unable to continue because a certain condition

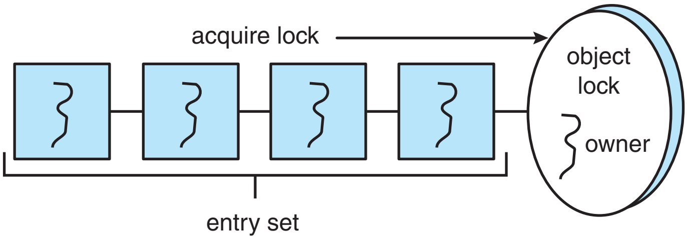


**Figure 7.10** Entry set for a lock.

### *BLOCK SYNCHRONIZATION*

The amount of time between when a lock is acquired and when it is released is defined as the **scope** of the lock. A `synchronized` method that has only a small percentage of its code manipulating shared data may yield a scope that is too large. In such an instance, it may be better to synchronize only the block of code that manipulates shared data than to synchronize the entire method. Such a design results in a smaller lock scope. Thus, in addition to declaring `synchronized` methods, Java also allows block synchronization, as illustrated below. Only the access to the critical-section code requires ownership of the object lock for the `this` object.

```
public void someMethod() {
   /* non-critical section */

   synchronized(this) {
      /* critical section */
   }

   /* remainder section */
}
```

has not been met. That will happen, for example, if the producer calls the `insert()` method and the buffer is full. The thread then will release the lock and wait until the condition that will allow it to continue is met.

When a thread calls the `wait()` method, the following happens:

1. The thread releases the lock for the object.
2. The state of the thread is set to blocked.
3. The thread is placed in the wait set for the object.

Consider the example in Figure 7.11. If the producer calls the `insert()` method and sees that the buffer is full, it calls the `wait()` method. This call releases the lock, blocks the producer, and puts the producer in the wait set for the object. Because the producer has released the lock, the consumer ultimately enters the `remove()` method, where it frees space in the buffer for the producer. Figure 7.12 illustrates the entry and wait sets for a lock. (Note that although `wait()` can throw an `InterruptedException`, we choose to ignore it for code clarity and simplicity.)

How does the consumer thread signal that the producer may now proceed? Ordinarily, when a thread exits a `synchronized` method, the departing thread releases only the lock associated with the object, possibly removing a thread from the entry set and giving it ownership of the lock. However, at the end of the `insert()` and `remove()` methods, we have a call to the method `notify()`. The call to `notify()`:

1. Picks an arbitrary thread `T` from the list of threads in the wait set

```
/* Producers call this method */
public synchronized void insert(E item) {
   while (count == BUFFER_SIZE) {
      try {
         wait();
      }
      catch (InterruptedException ie) { }
   }

   buffer[in] = item;
   in = (in + 1) % BUFFER_SIZE;
   count++;

   notify();
}

/* Consumers call this method */
public synchronized E remove() {
   E item;

   while (count == 0) {
      try {
         wait();
      }
      catch (InterruptedException ie) { }
   }

   item = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
   count--;

   notify();

   return item;
}
```

**Figure 7.11** `insert()` and `remove()` methods using `wait()` and `notify()`.

2. Moves T from the wait set to the entry set
3. Sets the state of T from blocked to runnable

T is now eligible to compete for the lock with the other threads. Once T has regained control of the lock, it returns from calling `wait()`, where it may check the value of `count` again. (Again, the selection of an *arbitrary* thread is according to the Java specification; in practice, most Java virtual machines order threads in the wait set according to a FIFO policy.)

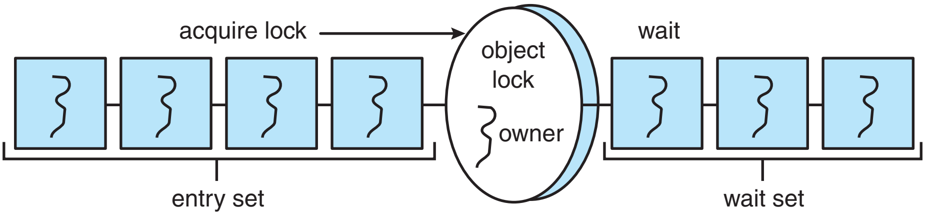


**Figure 7.12** Entry and wait sets.

Next, we describe the `wait()` and `notify()` methods in terms of the methods shown in Figure 7.11. We assume that the buffer is full and the lock for the object is available.

- The producer calls the `insert()` method, sees that the lock is available, and enters the method. Once in the method, the producer determines that the buffer is full and calls `wait()`. The call to `wait()` releases the lock for the object, sets the state of the producer to blocked, and puts the producer in the wait set for the object.
- The consumer ultimately calls and enters the `remove()` method, as the lock for the object is now available. The consumer removes an item from the buffer and calls `notify()`. Note that the consumer still owns the lock for the object.
- The call to `notify()` removes the producer from the wait set for the object, moves the producer to the entry set, and sets the producer's state to runnable.
- The consumer exits the `remove()` method. Exiting this method releases the lock for the object.
- The producer tries to reacquire the lock and is successful. It resumes execution from the call to `wait()`. The producer tests the `while` loop, determines that room is available in the buffer, and proceeds with the remainder of the `insert()` method. If no thread is in the wait set for the object, the call to `notify()` is ignored. When the producer exits the method, it releases the lock for the object.

The `synchronized`, `wait()`, and `notify()` mechanisms have been part of Java since its origins. However, later revisions of the Java API introduced much more flexible and robust locking mechanisms, some of which we examine in the following sections.

### 7.4.2 Reentrant Locks

Perhaps the simplest locking mechanism available in the API is the `ReentrantLock`. In many ways, a `ReentrantLock` acts like the `synchronized` statement described in Section 7.4.1: a `ReentrantLock` is owned by a single thread and is used to provide mutually exclusive access to a shared resource. However, the `ReentrantLock` provides several additional features, such as setting a *fairness* parameter, which favors granting the lock to the longest-waiting thread. (Recall that the specification for the JVM does not indicate that threads in the wait set for an object lock are to be ordered in any specific fashion.)

A thread acquires a `ReentrantLock` lock by invoking its `lock()` method. If the lock is available—or if the thread invoking `lock()` already owns it, which is why it is termed ***reentrant***—`lock()` assigns the invoking thread lock ownership and returns control. If the lock is unavailable, the invoking thread blocks until it is ultimately assigned the lock when its owner invokes `unlock()`. `ReentrantLock` implements the `Lock` interface; it is used as follows:

```
Lock key = new ReentrantLock();

key.lock();
try {
   /* critical section */
}
finally {
   key.unlock();
}
```

The programming idiom of using `try` and `finally` requires a bit of explanation. If the lock is acquired via the `lock()` method, it is important that the lock be similarly released. By enclosing `unlock()` in a `finally` clause, we ensure that the lock is released once the critical section completes or if an exception occurs within the `try` block. Notice that we do not place the call to `lock()` within the `try` clause, as `lock()` does not throw any checked exceptions. Consider what happens if we place `lock()` within the `try` clause and an unchecked exception occurs when `lock()` is invoked (such as `OutofMemoryError`): The `finally` clause triggers the call to `unlock()`, which then throws the unchecked `IllegalMonitorStateException`, as the lock was never acquired. This `IllegalMonitorStateException` replaces the unchecked exception that occurred when `lock()` was invoked, thereby obscuring the reason why the program initially failed.

Whereas a `ReentrantLock` provides mutual exclusion, it may be too conservative a strategy if multiple threads only read, but do not write, shared data. (We described this scenario in Section 7.1.2.) To address this need, the Java API also provides a `ReentrantReadWriteLock`, which is a lock that allows multiple concurrent readers but only one writer.

### 7.4.3 Semaphores

The Java API also provides a counting semaphore, as described in Section 6.6. The constructor for the semaphore appears as

```
Semaphore(int value);
```

where `value` specifies the initial value of the semaphore (a negative value is allowed). The `acquire()` method throws an `InterruptedException` if the acquiring thread is interrupted. The following example illustrates using a semaphore for mutual exclusion:

```
Semaphore sem = new Semaphore(1);

try {
   sem.acquire();
   /* critical section */
}
catch (InterruptedException ie) { }
finally {
   sem.release();
}
```

Notice that we place the call to `release()` in the `finally` clause to ensure that the semaphore is released.

### 7.4.4 Condition Variables

The last utility we cover in the Java API is the condition variable. Just as the `ReentrantLock` is similar to Java's `synchronized` statement, condition variables provide functionality similar to the `wait()` and `notify()` methods. Therefore, to provide mutual exclusion, a condition variable must be associated with a reentrant lock.

We create a condition variable by first creating a `ReentrantLock` and invoking its `newCondition()` method, which returns a `Condition` object representing the condition variable for the associated `ReentrantLock`. This is illustrated in the following statements:

```
Lock key = new ReentrantLock();
Condition condVar = key.newCondition();
```

Once the condition variable has been obtained, we can invoke its `await()` and `signal()` methods, which function in the same way as the `wait()` and `signal()` commands described in Section 6.7.

Recall that with monitors as described in Section 6.7, the `wait()` and `signal()` operations can be applied to ***named*** condition variables, allowing a thread to wait for a specific condition or to be notified when a specific condition has been met. At the language level, Java does not provide support for named condition variables. Each Java monitor is associated with just one unnamed condition variable, and the `wait()` and `notify()` operations described in Section 7.4.1 apply only to this single condition variable. When a Java thread is awakened via `notify()`, it receives no information as to why it was awakened; it is up to the reactivated thread to check for itself whether the condition for which it was waiting has been met. Condition variables remedy this by allowing a specific thread to be notified.

We illustrate with the following example: Suppose we have five threads, numbered 0 through 4, and a shared variable `turn` indicating which thread's turn it is. When a thread wishes to do work, it calls the `doWork()` method in Figure 7.13, passing its thread number. Only the thread whose value of `threadNumber` matches the value of `turn` can proceed; other threads must wait their turn.

```
/* threadNumber is the thread that wishes to do some work */
public void doWork(int threadNumber)
{
  lock.lock();

  try {
    /**
     * If it's not my turn, then wait
     * until I'm signaled.
     */
    if (threadNumber != turn)
      condVars[threadNumber].await();

    /**
     * Do some work for awhile ...
     */

    /**
     * Now signal to the next thread.
     */
    turn = (turn + 1) % 5;
    condVars[turn].signal();
  }
  catch (InterruptedException ie) { }
  finally {
    lock.unlock();
  }
}
```

**Figure 7.13** Example using Java condition variables.

We also must create a `ReentrantLock` and five condition variables (representing the conditions the threads are waiting for) to signal the thread whose turn is next. This is shown below:

```
Lock lock = new ReentrantLock();
Condition[] condVars = new Condition[5];

for (int i = 0; i < 5; i++)
  condVars[i] = lock.newCondition();
```

When a thread enters `doWork()`, it invokes the `await()` method on its associated condition variable if its `threadNumber` is not equal to `turn`, only to resume when it is signaled by another thread. After a thread has completed its work, it signals the condition variable associated with the thread whose turn follows.

It is important to note that `doWork()` does not need to be declared `synchronized`, as the `ReentrantLock` provides mutual exclusion. When a thread invokes `await()` on the condition variable, it releases the associated `ReentrantLock`, allowing another thread to acquire the mutual exclusion lock. Similarly, when `signal()` is invoked, only the condition variable is signaled; the lock is released by invoking `unlock()`.

## 7.5 Alternative Approaches

With the emergence of multicore systems has come increased pressure to develop concurrent applications that take advantage of multiple processing cores. However, concurrent applications present an increased risk of race conditions and liveness hazards such as deadlock. Traditionally, techniques such as mutex locks, semaphores, and monitors have been used to address these issues, but as the number of processing cores increases, it becomes increasingly difficult to design multithreaded applications that are free from race conditions and deadlock. In this section, we explore various features provided in both programming languages and hardware that support the design of thread-safe concurrent applications.

### 7.5.1 Transactional Memory

Quite often in computer science, ideas from one area of study can be used to solve problems in other areas. The concept of **transactional memory** originated in database theory, for example, yet it provides a strategy for process synchronization. A **memory transaction** is a sequence of memory read–write operations that are atomic. If all operations in a transaction are completed, the memory transaction is committed. Otherwise, the operations must be aborted and rolled back. The benefits of transactional memory can be obtained through features added to a programming language.

Consider an example. Suppose we have a function `update()` that modifies shared data. Traditionally, this function would be written using mutex locks (or semaphores) such as the following:

```
void update ()
{
   acquire();

   /* modify shared data */

   release();
}
```

However, using synchronization mechanisms such as mutex locks and semaphores involves many potential problems, including deadlock. Additionally, as the number of threads increases, traditional locking doesn't scale as well, because the level of contention among threads for lock ownership becomes very high.

As an alternative to traditional locking methods, new features that take advantage of transactional memory can be added to a programming language. In our example, suppose we add the construct `atomic{S}`, which ensures that the operations in `S` execute as a transaction. This allows us to rewrite the `update()` function as follows:

```
void update ()
{
   atomic {
      /* modify shared data */
   }
}
```

The advantage of using such a mechanism rather than locks is that the transactional memory system—not the developer—is responsible for guaranteeing atomicity. Additionally, because no locks are involved, deadlock is not possible. Furthermore, a transactional memory system can identify which statements in atomic blocks can be executed concurrently, such as concurrent read access to a shared variable. It is, of course, possible for a programmer to identify these situations and use reader–writer locks, but the task becomes increasingly difficult as the number of threads within an application grows.

Transactional memory can be implemented in either software or hardware. **Software transactional memory** (**STM**), as the name suggests, implements transactional memory exclusively in software—no special hardware is needed. STM works by inserting instrumentation code inside transaction blocks. The code is inserted by a compiler and manages each transaction by examining where statements may run concurrently and where specific low-level locking is required. **Hardware transactional memory** (**HTM**) uses hardware cache hierarchies and cache coherency protocols to manage and resolve conflicts involving shared data residing in separate processors' caches. HTM requires no special code instrumentation and thus has less overhead than STM. However, HTM does require that existing cache hierarchies and cache coherency protocols be modified to support transactional memory.

Transactional memory has existed for several years without widespread implementation. However, the growth of multicore systems and the associated emphasis on concurrent and parallel programming have prompted a significant amount of research in this area on the part of both academics and commercial software and hardware vendors.

### 7.5.2 OpenMP

In Section 4.5.2, we provided an overview of OpenMP and its support of parallel programming in a shared-memory environment. Recall that OpenMP includes a set of compiler directives and an API. Any code following the compiler directive `#pragma omp parallel` is identified as a parallel region and is performed by a number of threads equal to the number of processing cores in the system. The advantage of OpenMP (and similar tools) is that thread creation and management are handled by the OpenMP library and are not the responsibility of application developers.

Along with its `#pragma omp parallel` compiler directive, OpenMP provides the compiler directive `#pragma omp critical`, which specifies the code region following the directive as a critical section in which only one thread may be active at a time. In this way, OpenMP provides support for ensuring that threads do not generate race conditions.

As an example of the use of the critical-section compiler directive, first assume that the shared variable `counter` can be modified in the `update()` function as follows:

```
void update(int value)
{
    counter += value;
}
```

If the `update()` function can be part of—or invoked from—a parallel region, a race condition is possible on the variable `counter`.

The critical-section compiler directive can be used to remedy this race condition and is coded as follows:

```
void update(int value)
{
    #pragma omp critical
    {
        counter += value;
    }
}
```

The critical-section compiler directive behaves much like a binary semaphore or mutex lock, ensuring that only one thread at a time is active in the critical section. If a thread attempts to enter a critical section when another thread is currently active in that section (that is, ***owns*** the section), the calling thread is blocked until the owner thread exits. If multiple critical sections must be used, each critical section can be assigned a separate name, and a rule can specify that no more than one thread may be active in a critical section of the same name simultaneously.

An advantage of using the critical-section compiler directive in OpenMP is that it is generally considered easier to use than standard mutex locks. However, a disadvantage is that application developers must still identify possible race conditions and adequately protect shared data using the compiler directive. Additionally, because the critical-section compiler directive behaves much like a mutex lock, deadlock is still possible when two or more critical sections are identified.

### 7.5.3 Functional Programming Languages

Most well-known programming languages—such as C, C++, Java, and C#—are known as **imperative** (or **procedural**) languages. Imperative languages are used for implementing algorithms that are state-based. In these languages, the flow of the algorithm is crucial to its correct operation, and state is represented with variables and other data structures. Of course, program state is mutable, as variables may be assigned different values over time.

With the current emphasis on concurrent and parallel programming for multicore systems, there has been greater focus on **functional** programming languages, which follow a programming paradigm much different from that offered by imperative languages. The fundamental difference between imperative and functional languages is that functional languages do not maintain state. That is, once a variable has been defined and assigned a value, its value is immutable—it cannot change. Because functional languages disallow mutable state, they need not be concerned with issues such as race conditions and deadlocks. Essentially, most of the problems addressed in this chapter are nonexistent in functional languages.

Several functional languages are presently in use, and we briefly mention two of them here: Erlang and Scala. The Erlang language has gained significant attention because of its support for concurrency and the ease with which it can be used to develop applications that run on parallel systems. Scala is a functional language that is also object-oriented. In fact, much of the syntax of Scala is similar to the popular object-oriented languages Java and C#. Readers interested in Erlang and Scala, and in further details about functional languages in general, are encouraged to consult the bibliography at the end of this chapter for additional references.

## 7.6 Summary

- Classic problems of process synchronization include the bounded-buffer, readers–writers, and dining-philosophers problems. Solutions to these problems can be developed using the tools presented in Chapter 6, including mutex locks, semaphores, monitors, and condition variables.
- Windows uses dispatcher objects as well as events to implement process synchronization tools.
- Linux uses a variety of approaches to protect against race conditions, including atomic variables, spinlocks, and mutex locks.
- The POSIX API provides mutex locks, semaphores, and condition variables. POSIX provides two forms of semaphores: named and unnamed. Several unrelated processes can easily access the same named semaphore by simply referring to its name. Unnamed semaphores cannot be shared as easily, and require placing the semaphore in a region of shared memory.
- Java has a rich library and API for synchronization. Available tools include monitors (which are provided at the language level) as well as reentrant locks, semaphores, and condition variables (which are supported by the API).
- Alternative approaches to solving the critical-section problem include transactional memory, OpenMP, and functional languages. Functional languages are particularly intriguing, as they offer a different programming paradigm from procedural languages. Unlike procedural languages, functional languages do not maintain state and therefore are generally immune from race conditions and critical sections.

## Practice Exercises

**7.1** Explain why Windows and Linux implement multiple locking mechanisms. Describe the circumstances under which they use spinlocks, mutex locks, semaphores, and condition variables. In each case, explain why the mechanism is needed.

**7.2** Windows provides a lightweight synchronization tool called **slim reader–writer** locks. Whereas most implementations of reader–writer locks favor either readers or writers, or perhaps order waiting threads using a FIFO policy, slim reader–writer locks favor neither readers nor writers, nor are waiting threads ordered in a FIFO queue. Explain the benefits of providing such a synchronization tool.

**7.3** Describe what changes would be necessary to the producer and consumer processes in Figure 7.1 and Figure 7.2 so that a mutex lock could be used instead of a binary semaphore.

**7.4** Describe how deadlock is possible with the dining-philosophers problem.

**7.5** Explain the difference between signaled and non-signaled states with Windows dispatcher objects.

**7.6** Assume `val` is an atomic integer in a Linux system. What is the value of `val` after the following operations have been completed?

```
atomic_set(&val,10);
atomic_sub(8,&val);
atomic_inc(&val);
atomic_inc(&val);
atomic_add(6,&val);
atomic_sub(3,&val);
```

## Further Reading

Details of Windows synchronization can be found in [Solomon and Russinovich (2000)]. [Love (2010)] describes synchronization in the Linux kernel. [Hart (2005)] describes thread synchronization using Windows. [Breshears (2009)] and [Pacheco (2011)] provide detailed coverage of synchronization issues in relation to parallel programming. Details on using OpenMP can be found at http://openmp.org. Both [Oaks (2014)] and [Goetz et al. (2006)] contrast traditional synchronization and CAS-based strategies in Java.

## Bibliography

**[Breshears (2009)]** C. Breshears, *The Art of Concurrency*, O'Reilly & Associates (2009).

**[Goetz et al. (2006)]** B. Goetz, T. Peirls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea, *Java Concurrency in Practice*, Addison-Wesley (2006).

**[Hart (2005)]** J. M. Hart, *Windows System Programming*, Third Edition, Addison-Wesley (2005).

**[Love (2010)]** R. Love, *Linux Kernel Development*, Third Edition, Developer's Library (2010).

**[Oaks (2014)]** S. Oaks, *Java Performance—The Definitive Guide*, O'Reilly & Associates (2014).

**[Pacheco (2011)]** P. S. Pacheco, *An Introduction to Parallel Programming*, Morgan Kaufmann (2011).

**[Solomon and Russinovich (2000)]** D. A. Solomon and M. E. Russinovich, *Inside Microsoft Windows 2000,* Third Edition, Microsoft Press (2000).

## Chapter 7 Exercises

**7.7** Describe two kernel data structures in which race conditions are possible. Be sure to include a description of how a race condition can occur.

**7.8** The Linux kernel has a policy that a process cannot hold a spinlock while attempting to acquire a semaphore. Explain why this policy is in place.

**7.9** Design an algorithm for a bounded-buffer monitor in which the buffers (portions) are embedded within the monitor itself.

**7.10** The strict mutual exclusion within a monitor makes the bounded-buffer monitor of Exercise 7.14 mainly suitable for small portions.

a. Explain why this is true.

b. Design a new scheme that is suitable for larger portions.

**7.11** Discuss the tradeoff between fairness and throughput of operations in the readers–writers problem. Propose a method for solving the readers–writers problem without causing starvation.

**7.12** Explain why the call to the `lock()` method in a Java `ReentrantLock` is not placed in the `try` clause for exception handling, yet the call to the `unlock()` method is placed in a `finally` clause.

**7.13** Explain the difference between software and hardware transactional memory.

## Programming Problems

**7.14** Exercise 3.20 required you to design a PID manager that allocated a unique process identifier to each process. Exercise 4.28 required you to modify your solution to Exercise 3.20 by writing a program that created a number of threads that requested and released process identifiers. Using mutex locks, modify your solution to Exercise 4.28 by ensuring that the data structure used to represent the availability of process identifiers is safe from race conditions.

**7.15** In Exercise 4.27, you wrote a program to generate the Fibonacci sequence. The program required the parent thread to wait for the child thread to finish its execution before printing out the computed values. If we let the parent thread access the Fibonacci numbers as soon as they were computed by the child thread—rather than waiting for the child thread to terminate—what changes would be necessary to the solution for this exercise? Implement your modified solution.

**7.16** The C program `stack-ptr.c` (available in the source-code download) contains an implementation of a stack using a linked list. An example of its use is as follows:

```
StackNode *top = NULL;
push(5, &top);
push(10, &top);
push(15, &top);

int value = pop(&top);
value = pop(&top);
value = pop(&top);
```

This program currently has a race condition and is not appropriate for a concurrent environment. Using Pthreads mutex locks (described in Section 7.3.1), fix the race condition.

**7.17** Exercise 4.24 asked you to design a multithreaded program that estimated $\pi$ using the Monte Carlo technique. In that exercise, you were asked to create a single thread that generated random points, storing the result in a global variable. Once that thread exited, the parent thread performed the calculation that estimated the value of $\pi$. Modify that program so that you create several threads, each of which generates random points and determines if the points fall within the circle. Each thread will have to update the global count of all points that fall within the circle. Protect against race conditions on updates to the shared global variable by using mutex locks.

**7.18** Exercise 4.25 asked you to design a program using OpenMP that estimated $\pi$ using the Monte Carlo technique. Examine your solution to that program looking for any possible race conditions. If you identify a race condition, protect against it using the strategy outlined in Section 7.5.2.

**7.19** A **barrier** is a tool for synchronizing the activity of a number of threads. When a thread reaches a **barrier point**, it cannot proceed until all other threads have reached this point as well. When the last thread reaches the barrier point, all threads are released and can resume concurrent execution.

Assume that the barrier is initialized to $N$—the number of threads that must wait at the barrier point:

```
init(N);
```

Each thread then performs some work until it reaches the barrier point:

```
/* do some work for awhile */

barrier_point();

/* do some work for awhile */
```

Using either the POSIX or Java synchronization tools described in this chapter, construct a barrier that implements the following API:

- `int init(int n)`—Initializes the barrier to the specified size.
- `int barrier_point(void)`—Identifies the barrier point. All threads are released from the barrier when the last thread reaches this point.

The return value of each function is used to identify error conditions. Each function will return 0 under normal operation and will return −1 if an error occurs. A testing harness is provided in the source-code download to test your implementation of the barrier.

## Programming Projects

### Project 1—Designing a Thread Pool

Thread pools were introduced in Section 4.5.1. When thread pools are used, a task is submitted to the pool and executed by a thread from the pool. Work is submitted to the pool using a queue, and an available thread removes work from the queue. If there are no available threads, the work remains queued until one becomes available. If there is no work, threads await notification until a task becomes available.

This project involves creating and managing a thread pool, and it may be completed using either Pthreds and POSIX synchronization or Java. Below we provide the details relevant to each specific technology.

### I. POSIX

The POSIX version of this project will involve creating a number of threads using the Pthreads API as well as using POSIX mutex locks and semaphores for synchronization.

### The Client

Users of the thread pool will utilize the following API:

- `void pool_init()`—Initializes the thread pool.
- `int pool_submit(void (*somefunction)(void *p), void *p)`— where `somefunction` is a pointer to the function that will be executed by a thread from the pool and `p` is a parameter passed to the function.
- `void pool_shutdown(void)`—Shuts down the thread pool once all tasks have completed.

We provide an example program `client.c` in the source code download that illustrates how to use the thread pool using these functions.

### Implementation of the Thread Pool

In the source code download we provide the C source file `threadpool.c` as a partial implementation of the thread pool. You will need to implement the functions that are called by client users, as well as several additional functions that support the internals of the thread pool. Implementation will involve the following activities:

1. The `pool_init()` function will create the threads at startup as well as initialize mutual-exclusion locks and semaphores.
2. The `pool_submit()` function is partially implemented and currently places the function to be executed—as well as its data— into a `task` struct. The `task` struct represents work that will be completed by a thread in the pool. `pool_submit()` will add these tasks to the queue by invoking the `enqueue()` function, and worker threads will call `dequeue()` to retrieve work from the queue. The queue may be implemented statically (using arrays) or dynamically (using a linked list).

   The `pool_init()` function has an `int` return value that is used to indicate if the task was successfully submitted to the pool (0 indicates success, 1 indicates failure). If the queue is implemented using arrays, `pool_init()` will return 1 if there is an attempt to submit work and the queue is full. If the queue is implemented as a linked list, `pool_init()` should always return 0 unless a memory allocation error occurs.
3. The `worker()` function is executed by each thread in the pool, where each thread will wait for available work. Once work becomes available, the thread will remove it from the queue and invoke `execute()` to run the specified function.

   A semaphore can be used for notifying a waiting thread when work is submitted to the thread pool. Either named or unnamed semaphores may be used. Refer to Section 7.3.2 for further details on using POSIX semaphores.

4. A mutex lock is necessary to avoid race conditions when accessing or modifying the queue. (Section 7.3.1 provides details on Pthreads mutex locks.)
5. The `pool_shutdown()` function will cancel each worker thread and then wait for each thread to terminate by calling `pthread_join()`. Refer to Section 4.6.3 for details on POSIX thread cancellation. (The semaphore operation `sem_wait()` is a cancellation point that allows a thread waiting on a semaphore to be cancelled.)

Refer to the source-code download for additional details on this project. In particular, the `README` file describes the source and header files, as well as the `Makefile` for building the project.

## II. Java

The Java version of this project may be completed using Java synchronization tools as described in Section 7.4. Synchronization may depend on either (a) monitors using `synchronized/wait()/notify()` (Section 7.4.1) or (b) semaphores and reentrant locks (Section 7.4.2 and Section 7.4.3). Java threads are described in Section 4.4.3.

### Implementation of the Thread Pool

Your thread pool will implement the following API:

- `ThreadPool()`—Create a default-sized thread pool.
- `ThreadPool(int size)`—Create a thread pool of size `size`.
- `void add(Runnable task)`—Add a task to be performed by a thread in the pool.
- `void shutdown()`—Stop all threads in the pool.

We provide the Java source file `ThreadPool.java` as a partial implementation of the thread pool in the source code download. You will need to implement the methods that are called by client users, as well as several additional methods that support the internals of the thread pool. Implementation will involve the following activities:

1. The constructor will first create a number of idle threads that await work.
2. Work will be submitted to the pool via the `add()` method, which adds a task implementing the `Runnable` interface. The `add()` method will place the `Runnable` task into a queue (you may use an available structure from the Java API such as `java.util.List`).
3. Once a thread in the pool becomes available for work, it will check the queue for any `Runnable` tasks. If there is such a task, the idle thread will remove the task from the queue and invoke its `run()` method. If the queue is empty, the idle thread will wait to be notified when work

becomes available. (The `add()` method may implement notification using either `notify()` or semaphore operations when it places a `Runnable` task into the queue to possibly awaken an idle thread awaiting work.)

4. The `shutdown()` method will stop all threads in the pool by invoking their `interrupt()` method. This, of course, requires that `Runnable` tasks being executed by the thread pool check their interruption status (Section 4.6.3).

Refer to the source-code download for additional details on this project. In particular, the README file describes the Java source files, as well as further details on Java thread interruption.

## Project 2—The Sleeping Teaching Assistant

A university computer science department has a teaching assistant (TA) who helps undergraduate students with their programming assignments during regular office hours. The TA's office is rather small and has room for only one desk with a chair and computer. There are three chairs in the hallway outside the office where students can sit and wait if the TA is currently helping another student. When there are no students who need help during office hours, the TA sits at the desk and takes a nap. If a student arrives during office hours and finds the TA sleeping, the student must awaken the TA to ask for help. If a student arrives and finds the TA currently helping another student, the student sits on one of the chairs in the hallway and waits. If no chairs are available, the student will come back at a later time.

Using POSIX threads, mutex locks, and semaphores, implement a solution that coordinates the activities of the TA and the students. Details for this assignment are provided below.

### The Students and the TA

Using Pthreads (Section 4.4.1), begin by creating *n* students where each student will run as a separate thread. The TA will run as a separate thread as well. Student threads will alternate between programming for a period of time and seeking help from the TA. If the TA is available, they will obtain help. Otherwise, they will either sit in a chair in the hallway or, if no chairs are available, will resume programming and will seek help at a later time. If a student arrives and notices that the TA is sleeping, the student must notify the TA using a semaphore. When the TA finishes helping a student, the TA must check to see if there are students waiting for help in the hallway. If so, the TA must help each of these students in turn. If no students are present, the TA may return to napping.

Perhaps the best option for simulating students programming—as well as the TA providing help to a student—is to have the appropriate threads sleep for a random period of time.

Coverage of POSIX mutex locks and semaphores is provided in Section 7.3. Consult that section for details.

## Project 3—The Dining-Philosophers Problem

In Section 7.1.3, we provide an outline of a solution to the dining-philosophers problem using monitors. This project involves implementing a solution to this problem using either POSIX mutex locks and condition variables or Java condition variables. Solutions will be based on the algorithm illustrated in Figure 7.7.

Both implementations will require creating five philosophers, each identified by a number 0 . . 4. Each philosopher will run as a separate thread. Philosophers alternate between thinking and eating. To simulate both activities, have each thread sleep for a random period between one and three seconds.

### I. POSIX

Thread creation using Pthreads is covered in Section 4.4.1. When a philosopher wishes to eat, she invokes the function

```
pickup_forks(int philosopher_number)
```

where `philosopher_number` identifies the number of the philosopher wishing to eat. When a philosopher finishes eating, she invokes

```
return_forks(int philosopher_number)
```

Your implementation will require the use of POSIX condition variables, which are covered in Section 7.3.

### II. Java

When a philosopher wishes to eat, she invokes the method `takeForks(philosopherNumber)`, where `philosopherNumber` identifies the number of the philosopher wishing to eat. When a philosopher finishes eating, she invokes `returnForks(philosopherNumber)`.

Your solution will implement the following interface:

```
public interface DiningServer
{
  /* Called by a philosopher when it wishes to eat */
  public void takeForks(int philosopherNumber);

  /* Called by a philosopher when it is finished eating */
  public void returnForks(int philosopherNumber);
}
```

It will require the use of Java condition variables, which are covered in Section 7.4.4.

# Project 4—The Producer–Consumer Problem

In Section 7.1.1, we presented a semaphore-based solution to the producer–consumer problem using a bounded buffer. In this project, you will design a programming solution to the bounded-buffer problem using the producer and consumer processes shown in Figures 5.9 and 5.10. The solution presented in Section 7.1.1 uses three semaphores: `empty` and `full`, which count the number of empty and full slots in the buffer, and `mutex`, which is a binary (or mutual-exclusion) semaphore that protects the actual insertion or removal of items in the buffer. For this project, you will use standard counting semaphores for `empty` and `full` and a mutex lock, rather than a binary semaphore, to represent `mutex`. The producer and consumer—running as separate threads—will move items to and from a buffer that is synchronized with the `empty`, `full`, and `mutex` structures. You can solve this problem using either Pthreads or the Windows API.

## The Buffer

Internally, the buffer will consist of a fixed-size array of type `buffer_item` (which will be defined using a `typedef`). The array of `buffer_item` objects will be manipulated as a circular queue. The definition of `buffer_item`, along with the size of the buffer, can be stored in a header file such as the following:

```
/* buffer.h */
typedef int buffer_item;
#define BUFFER_SIZE 5
```

The buffer will be manipulated with two functions, `insert_item()` and `remove_item()`, which are called by the producer and consumer threads, respectively. A skeleton outlining these functions appears in Figure 7.14.

The `insert_item()` and `remove_item()` functions will synchronize the producer and consumer using the algorithms outlined in Figure 7.1 and Figure 7.2. The buffer will also require an initialization function that initializes the mutual-exclusion object `mutex` along with the `empty` and `full` semaphores.

The `main()` function will initialize the buffer and create the separate producer and consumer threads. Once it has created the producer and consumer threads, the `main()` function will sleep for a period of time and, upon awakening, will terminate the application. The `main()` function will be passed three parameters on the command line:

1. How long to sleep before terminating
2. The number of producer threads
3. The number of consumer threads

A skeleton for this function appears in Figure 7.15.

```
#include "buffer.h"

/* the buffer */
buffer_item buffer[BUFFER_SIZE];

int insert_item(buffer_item item) {
   /* insert item into buffer
    return 0 if successful, otherwise
    return -1 indicating an error condition */
}

int remove_item(buffer_item *item) {
   /* remove an object from buffer
    placing it in item
    return 0 if successful, otherwise
    return -1 indicating an error condition */
}
```

**Figure 7.14** Outline of buffer operations.

### The Producer and Consumer Threads

The producer thread will alternate between sleeping for a random period of time and inserting a random integer into the buffer. Random numbers will be produced using the `rand()` function, which produces random integers between 0 and RAND_MAX. The consumer will also sleep for a random period of time and, upon awakening, will attempt to remove an item from the buffer. An outline of the producer and consumer threads appears in Figure 7.16.

```
#include "buffer.h"

int main(int argc, char *argv[]) {
 /* 1. Get command line arguments argv[1],argv[2],argv[3] */
 /* 2. Initialize buffer */
 /* 3. Create producer thread(s) */
 /* 4. Create consumer thread(s) */
 /* 5. Sleep */
 /* 6. Exit */
}
```

**Figure 7.15** Outline of skeleton program.

```
#include <stdlib.h> /* required for rand() */
#include "buffer.h"

void *producer(void *param) {
  buffer_item item;

  while (true) {
    /* sleep for a random period of time */
    sleep(...);
    /* generate a random number */
    item = rand();
    if (insert_item(item))
      fprintf("report error condition");
    else
      printf("producer produced %d\n",item);
}

void *consumer(void *param) {
  buffer_item item;

  while (true) {
    /* sleep for a random period of time */
    sleep(...);
    if (remove_item(&item))
      fprintf("report error condition");
    else
      printf("consumer consumed %d\n",item);
}
```

**Figure 7.16** An outline of the producer and consumer threads.

As noted earlier, you can solve this problem using either Pthreads or the Windows API. In the following sections, we supply more information on each of these choices.

### Pthreads Thread Creation and Synchronization

Creating threads using the Pthreads API is discussed in Section 4.4.1. Coverage of mutex locks and semaphores using Pthreads is provided in Section 7.3. Refer to those sections for specific instructions on Pthreads thread creation and synchronization.

### Windows Threads

Section 4.4.2 discusses thread creation using the Windows API. Refer to that section for specific instructions on creating threads.

### Windows Mutex Locks

Mutex locks are a type of dispatcher object, as described in Section 7.2.1. The following illustrates how to create a mutex lock using the `CreateMutex()` function:

```
#include <windows.h>

HANDLE Mutex;
Mutex = CreateMutex(NULL, FALSE, NULL);
```

The first parameter refers to a security attribute for the mutex lock. By setting this attribute to `NULL`, we prevent any children of the process from creating this mutex lock to inherit the handle of the lock. The second parameter indicates whether the creator of the mutex lock is the lock's initial owner. Passing a value of `FALSE` indicates that the thread creating the mutex is not the initial owner. (We shall soon see how mutex locks are acquired.) The third parameter allows us to name the mutex. However, because we provide a value of `NULL`, we do not name the mutex. If successful, `CreateMutex()` returns a `HANDLE` to the mutex lock; otherwise, it returns `NULL`.

In Section 7.2.1, we identified dispatcher objects as being either ***signaled*** or ***nonsignaled***. A signaled dispatcher object (such as a mutex lock) is available for ownership. Once it is acquired, it moves to the nonsignaled state. When it is released, it returns to signaled.

Mutex locks are acquired by invoking the `WaitForSingleObject()` function. The function is passed the `HANDLE` to the lock along with a flag indicating how long to wait. The following code demonstrates how the mutex lock created above can be acquired:

```
WaitForSingleObject(Mutex, INFINITE);
```

The parameter value `INFINITE` indicates that we will wait an infinite amount of time for the lock to become available. Other values could be used that would allow the calling thread to time out if the lock did not become available within a specified time. If the lock is in a signaled state, `WaitForSingleObject()` returns immediately, and the lock becomes nonsignaled. A lock is released (moves to the signaled state) by invoking `ReleaseMutex()`—for example, as follows:

```
ReleaseMutex(Mutex);
```

### Windows Semaphores

Semaphores in the Windows API are dispatcher objects and thus use the same signaling mechanism as mutex locks. Semaphores are created as follows:

```
#include <windows.h>

HANDLE Sem;
Sem = CreateSemaphore(NULL, 1, 5, NULL);
```

The first and last parameters identify a security attribute and a name for the semaphore, similar to what we described for mutex locks. The second and third parameters indicate the initial value and maximum value of the semaphore. In this instance, the initial value of the semaphore is 1, and its maximum value is 5. If successful, `CreateSemaphore()` returns a `HANDLE` to the mutex lock; otherwise, it returns `NULL`.

Semaphores are acquired with the same `WaitForSingleObject()` function as mutex locks. We acquire the semaphore `Sem` created in this example by using the following statement:

```
WaitForSingleObject(Sem, INFINITE);
```

If the value of the semaphore is $> 0$, the semaphore is in the signaled state and thus is acquired by the calling thread. Otherwise, the calling thread blocks indefinitely—as we are specifying `INFINITE`—until the semaphore returns to the signaled state.

The equivalent of the `signal()` operation for Windows semaphores is the `ReleaseSemaphore()` function. This function is passed three parameters:

1. The HANDLE of the semaphore
2. How much to increase the value of the semaphore
3. A pointer to the previous value of the semaphore

We can use the following statement to increase `Sem` by 1:

```
ReleaseSemaphore(Sem, 1, NULL);
```

Both `ReleaseSemaphore()` and `ReleaseMutex()` return a nonzero value if successful and 0 otherwise.

# CHAPTER 8

# *Deadlocks*

In a multiprogramming environment, several threads may compete for a finite number of resources. A thread requests resources; if the resources are not available at that time, the thread enters a waiting state. Sometimes, a waiting thread can never again change state, because the resources it has requested are held by other waiting threads. This situation is called a **deadlock**. We discussed this issue briefly in Chapter 6 as a form of liveness failure. There, we defined deadlock as a situation in which *every process in a set of processes is waiting for an event that can be caused only by another process in the set*.

Perhaps the best illustration of a deadlock can be drawn from a law passed by the Kansas legislature early in the 20th century. It said, in part: "When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone."

In this chapter, we describe methods that application developers as well as operating-system programmers can use to prevent or deal with deadlocks. Although some applications can identify programs that may deadlock, operating systems typically do not provide deadlock-prevention facilities, and it remains the responsibility of programmers to ensure that they design deadlock-free programs. Deadlock problems—as well as other liveness failures—are becoming more challenging as demand continues for increased concurrency and parallelism on multicore systems.

## CHAPTER OBJECTIVES

- Illustrate how deadlock can occur when mutex locks are used.
- Define the four necessary conditions that characterize deadlock.
- Identify a deadlock situation in a resource allocation graph.
- Evaluate the four different approaches for preventing deadlocks.
- Apply the banker's algorithm for deadlock avoidance.
- Apply the deadlock detection algorithm.
- Evaluate approaches for recovering from deadlock.

## 8.1 System Model

A system consists of a finite number of resources to be distributed among a number of competing threads. The resources may be partitioned into several types (or classes), each consisting of some number of identical instances. CPU cycles, files, and I/O devices (such as network interfaces and DVD drives) are examples of resource types. If a system has four CPUs, then the resource type *CPU* has four instances. Similarly, the resource type *network* may have two instances. If a thread requests an instance of a resource type, the allocation of ***any*** instance of the type should satisfy the request. If it does not, then the instances are not identical, and the resource type classes have not been defined properly.

The various synchronization tools discussed in Chapter 6, such as mutex locks and semaphores, are also system resources; and on contemporary computer systems, they are the most common sources of deadlock. However, definition is not a problem here. A lock is typically associated with a specific data structure—that is, one lock may be used to protect access to a queue, another to protect access to a linked list, and so forth. For that reason, each instance of a lock is typically assigned its own resource class.

Note that throughout this chapter we discuss kernel resources, but threads may use resources from other processes (for example, via interprocess communication), and those resource uses can also result in deadlock. Such deadlocks are not the concern of the kernel and thus not described here.

A thread must request a resource before using it and must release the resource after using it. A thread may request as many resources as it requires to carry out its designated task. Obviously, the number of resources requested may not exceed the total number of resources available in the system. In other words, a thread cannot request two network interfaces if the system has only one.

Under the normal mode of operation, a thread may utilize a resource in only the following sequence:

1. **Request**. The thread requests the resource. If the request cannot be granted immediately (for example, if a mutex lock is currently held by another thread), then the requesting thread must wait until it can acquire the resource.
2. **Use**. The thread can operate on the resource (for example, if the resource is a mutex lock, the thread can access its critical section).
3. **Release**. The thread releases the resource.

The request and release of resources may be system calls, as explained in Chapter 2. Examples are the `request()` and `release()` of a device, `open()` and `close()` of a file, and `allocate()` and `free()` memory system calls. Similarly, as we saw in Chapter 6, request and release can be accomplished through the `wait()` and `signal()` operations on semaphores and through `acquire()` and `release()` of a mutex lock. For each use of a kernel-managed resource by a thread, the operating system checks to make sure that the thread has requested and has been allocated the resource. A system table records whether each resource is free or allocated. For each resource that is allocated, the table also records the thread to which it is allocated. If a thread requests a resource that is currently allocated to another thread, it can be added to a queue of threads waiting for this resource.

A set of threads is in a deadlocked state when every thread in the set is waiting for an event that can be caused only by another thread in the set. The events with which we are mainly concerned here are resource acquisition and release. The resources are typically logical (for example, mutex locks, semaphores, and files); however, other types of events may result in deadlocks, including reading from a network interface or the IPC (interprocess communication) facilities discussed in Chapter 3.

To illustrate a deadlocked state, we refer back to the dining-philosophers problem from Section 7.1.3. In this situation, resources are represented by chopsticks. If all the philosophers get hungry at the same time, and each philosopher grabs the chopstick on her left, there are no longer any available chopsticks. Each philosopher is then blocked waiting for her right chopstick to become available.

Developers of multithreaded applications must remain aware of the possibility of deadlocks. The locking tools presented in Chapter 6 are designed to avoid race conditions. However, in using these tools, developers must pay careful attention to how locks are acquired and released. Otherwise, deadlock can occur, as described next.

## 8.2 Deadlock in Multithreaded Applications

Prior to examining how deadlock issues can be identified and managed, we first illustrate how deadlock can occur in a multithreaded Pthread program using POSIX mutex locks. The `pthread_mutex_init()` function initializes an unlocked mutex. Mutex locks are acquired and released using `pthread_mutex_lock()` and `pthread_mutex_unlock()`, respectively. If a thread attempts to acquire a locked mutex, the call to `pthread_mutex_lock()` blocks the thread until the owner of the mutex lock invokes `pthread_mutex_unlock()`.

Two mutex locks are created and initialized in the following code example:

```
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;

pthread_mutex_init(&first_mutex,NULL);
pthread_mutex_init(&second_mutex,NULL);
```

Next, two threads—`thread_one` and `thread_two`—are created, and both these threads have access to both mutex locks. `thread_one` and `thread_two` run in the functions `do_work_one()` and `do_work_two()`, respectively, as shown in Figure 8.1.

In this example, `thread_one` attempts to acquire the mutex locks in the order (1) `first_mutex`, (2) `second_mutex`. At the same time, `thread_two` attempts to acquire the mutex locks in the order (1) `second_mutex`, (2) `first_mutex`. Deadlock is possible if `thread_one` acquires `first_mutex` while `thread_two` acquires `second_mutex`.

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);

   pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);

   pthread_exit(0);
}
```

**Figure 8.1** Deadlock example.

Note that, even though deadlock is possible, it will not occur if `thread_one` can acquire and release the mutex locks for `first_mutex` and `second_mutex` before `thread_two` attempts to acquire the locks. And, of course, the order in which the threads run depends on how they are scheduled by the CPU scheduler. This example illustrates a problem with handling deadlocks: it is difficult to identify and test for deadlocks that may occur only under certain scheduling circumstances.

### 8.2.1 Livelock

**Livelock** is another form of liveness failure. It is similar to deadlock; both prevent two or more threads from proceeding, but the threads are unable to proceed for different reasons. Whereas deadlock occurs when every thread in a set is blocked waiting for an event that can be caused only by another thread in the set, livelock occurs when a thread continuously attempts an action that fails. Livelock is similar to what sometimes happens when two people attempt to pass in a hallway: One moves to his right, the other to her left, still obstructing each other's progress. Then he moves to his left, and she moves to her right, and so forth. They aren't blocked, but they aren't making any progress.

Livelock can be illustrated with the Pthreads `pthread_mutex_trylock()` function, which attempts to acquire a mutex lock without blocking. The code example in Figure 8.2 rewrites the example from Figure 8.1 so that it now uses `pthread_mutex_trylock()`. This situation can lead to livelock if `thread_one` acquires `first_mutex`, followed by `thread_two` acquiring `second_mutex`. Each thread then invokes `pthread_mutex_trylock()`, which fails, releases their respective locks, and repeats the same actions indefinitely.

Livelock typically occurs when threads retry failing operations at the same time. It thus can generally be avoided by having each thread retry the failing operation at random times. This is precisely the approach taken by Ethernet networks when a network collision occurs. Rather than trying to retransmit a packet immediately after a collision occurs, a host involved in a collision will ***backoff*** a random period of time before attempting to transmit again.

Livelock is less common than deadlock but nonetheless is a challenging issue in designing concurrent applications, and like deadlock, it may only occur under specific scheduling circumstances.

## 8.3 Deadlock Characterization

In the previous section we illustrated how deadlock could occur in multithreaded programming using mutex locks. We now look more closely at conditions that characterize deadlock.

### 8.3.1 Necessary Conditions

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. **Mutual exclusion**. At least one resource must be held in a nonsharable mode; that is, only one thread at a time can use the resource. If another thread requests that resource, the requesting thread must be delayed until the resource has been released.
2. **Hold and wait**. A thread must be holding at least one resource and waiting to acquire additional resources that are currently being held by other threads.
3. **No preemption**. Resources cannot be preempted; that is, a resource can be released only voluntarily by the thread holding it, after that thread has completed its task.
4. **Circular wait**. A set $\{T_0, T_1, ..., T_n\}$ of waiting threads must exist such that $T_0$ is waiting for a resource held by $T_1$, $T_1$ is waiting for a resource held by $T_2$, ..., $T_{n-1}$ is waiting for a resource held by $T_n$, and $T_n$ is waiting for a resource held by $T_0$.

We emphasize that all four conditions must hold for a deadlock to occur. The circular-wait condition implies the hold-and-wait condition, so the four

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   int done = 0;

   while (!done) {
      pthread_mutex_lock(&first_mutex);
      if (pthread_mutex_trylock(&second_mutex)) {
         /**
          * Do some work
          */
         pthread_mutex_unlock(&second_mutex);
         pthread_mutex_unlock(&first_mutex);
         done = 1;
      }
      else
         pthread_mutex_unlock(&first_mutex);
   }

   pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
   int done = 0;

   while (!done) {
      pthread_mutex_lock(&second_mutex);
      if (pthread_mutex_trylock(&first_mutex)) {
         /**
          * Do some work
          */
         pthread_mutex_unlock(&first_mutex);
         pthread_mutex_unlock(&second_mutex);
         done = 1;
      }
      else
         pthread_mutex_unlock(&second_mutex);
   }

   pthread_exit(0);
}
```

**Figure 8.2** Livelock example.

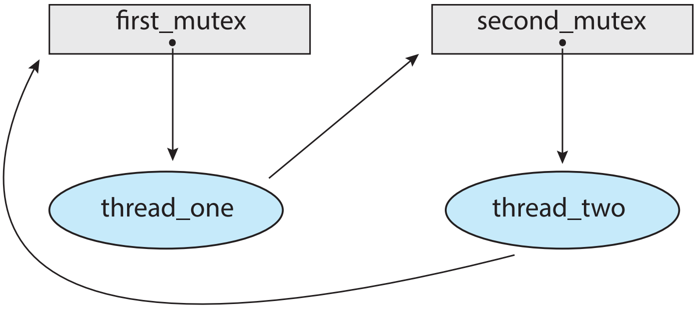


**Figure 8.3** Resource-allocation graph for program in Figure 8.1.

conditions are not completely independent. We shall see in Section 8.5, however, that it is useful to consider each condition separately.

### 8.3.2 Resource-Allocation Graph

Deadlocks can be described more precisely in terms of a directed graph called a **system resource-allocation graph**. This graph consists of a set of vertices $V$ and a set of edges $E$. The set of vertices $V$ is partitioned into two different types of nodes: $T = \{T_1, T_2, ..., T_n\}$, the set consisting of all the active threads in the system, and $R = \{R_1, R_2, ..., R_m\}$, the set consisting of all resource types in the system.

A directed edge from thread $T_i$ to resource type $R_j$ is denoted by $T_i \rightarrow R_j$; it signifies that thread $T_i$ has requested an instance of resource type $R_j$ and is currently waiting for that resource. A directed edge from resource type $R_j$ to thread $T_i$ is denoted by $R_j \rightarrow T_i$; it signifies that an instance of resource type $R_j$ has been allocated to thread $T_i$. A directed edge $T_i \rightarrow R_j$ is called a **request edge**; a directed edge $R_j \rightarrow T_i$ is called an **assignment edge**.

Pictorially, we represent each thread $T_i$ as a circle and each resource type $R_j$ as a rectangle. As a simple example, the resource allocation graph shown in Figure 8.3 illustrates the deadlock situation from the program in Figure 8.1. Since resource type $R_j$ may have more than one instance, we represent each such instance as a dot within the rectangle. Note that a request edge points only to the rectangle $R_j$, whereas an assignment edge must also designate one of the dots in the rectangle.

When thread $T_i$ requests an instance of resource type $R_j$, a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is ***instantaneously*** transformed to an assignment edge. When the thread no longer needs access to the resource, it releases the resource. As a result, the assignment edge is deleted.

The resource-allocation graph shown in Figure 8.4 depicts the following situation.

- The sets $T$, $R$, and $E$:
  - $T = \{T_1, T_2, T_3\}$
  - $R = \{R_1, R_2, R_3, R_4\}$

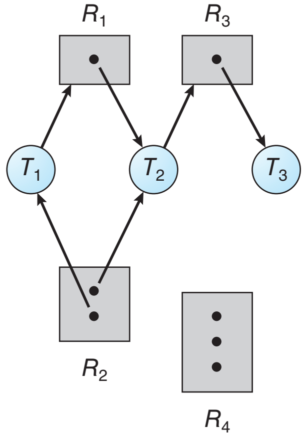


**Figure 8.4** Resource-allocation graph.

  - $E = \{T_1 \rightarrow R_1, T_2 \rightarrow R_3, R_1 \rightarrow T_2, R_2 \rightarrow T_2, R_2 \rightarrow T_1, R_3 \rightarrow T_3\}$

- Resource instances:
  - One instance of resource type $R_1$
  - Two instances of resource type $R_2$
  - One instance of resource type $R_3$
  - Three instances of resource type $R_4$
- Thread states:
  - Thread $T_1$ is holding an instance of resource type $R_2$ and is waiting for an instance of resource type $R_1$.
  - Thread $T_2$ is holding an instance of $R_1$ and an instance of $R_2$ and is waiting for an instance of $R_3$.
  - Thread $T_3$ is holding an instance of $R_3$.

Given the definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no thread in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. Each thread involved in the cycle is deadlocked. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.

If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

To illustrate this concept, we return to the resource-allocation graph depicted in Figure 8.4. Suppose that thread $T_3$ requests an instance of resource type $R_2$. Since no resource instance is currently available, we add a request

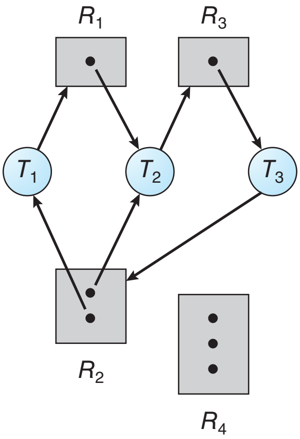


**Figure 8.5** Resource-allocation graph with a deadlock.

edge $T_3 \rightarrow R_2$ to the graph (Figure 8.5). At this point, two minimal cycles exist in the system:

$$T_1 \rightarrow R_1 \rightarrow T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$$
$$T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_2$$

Threads $T_1$, $T_2$, and $T_3$ are deadlocked. Thread $T_2$ is waiting for the resource $R_3$, which is held by thread $T_3$. Thread $T_3$ is waiting for either thread $T_1$ or thread $T_2$ to release resource $R_2$. In addition, thread $T_1$ is waiting for thread $T_2$ to release resource $R_1$.

Now consider the resource-allocation graph in Figure 8.6. In this example, we also have a cycle:

$$T_1 \rightarrow R_1 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$$

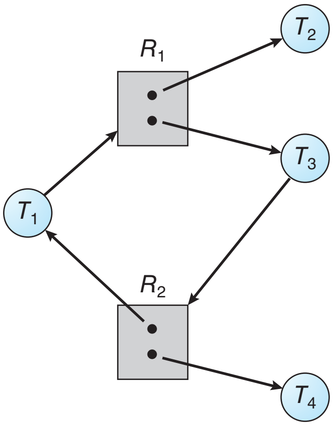


**Figure 8.6** Resource-allocation graph with a cycle but no deadlock.

However, there is no deadlock. Observe that thread $T_4$ may release its instance of resource type $R_2$. That resource can then be allocated to $T_3$, breaking the cycle.

In summary, if a resource-allocation graph does not have a cycle, then the system is ***not*** in a deadlocked state. If there is a cycle, then the system ***may*** or ***may not*** be in a deadlocked state. This observation is important when we deal with the deadlock problem.

## 8.4 Methods for Handling Deadlocks

Generally speaking, we can deal with the deadlock problem in one of three ways:

- We can ignore the problem altogether and pretend that deadlocks never occur in the system.
- We can use a protocol to prevent or avoid deadlocks, ensuring that the system will ***never*** enter a deadlocked state.
- We can allow the system to enter a deadlocked state, detect it, and recover.

The first solution is the one used by most operating systems, including Linux and Windows. It is then up to kernel and application developers to write programs that handle deadlocks, typically using approaches outlined in the second solution. Some systems—such as databases—adopt the third solution, allowing deadlocks to occur and then managing the recovery.

Next, we elaborate briefly on the three methods for handling deadlocks. Then, in Section 8.5 through Section 8.8, we present detailed algorithms. Before proceeding, we should mention that some researchers have argued that none of the basic approaches alone is appropriate for the entire spectrum of resource-allocation problems in operating systems. The basic approaches can be combined, however, allowing us to select an optimal approach for each class of resources in a system.

To ensure that deadlocks never occur, the system can use either a deadlock-prevention or a deadlock-avoidance scheme. **Deadlock prevention** provides a set of methods to ensure that at least one of the necessary conditions (Section 8.3.1) cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made. We discuss these methods in Section 8.5.

**Deadlock avoidance** requires that the operating system be given additional information in advance concerning which resources a thread will request and use during its lifetime. With this additional knowledge, the operating system can decide for each request whether or not the thread should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each thread, and the future requests and releases of each thread. We discuss these schemes in Section 8.6.

If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm, then a deadlock situation may arise. In this environment, the system can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred and an algorithm to recover from the deadlock (if a deadlock has indeed occurred). We discuss these issues in Section 8.7 and Section 8.8.

In the absence of algorithms to detect and recover from deadlocks, we may arrive at a situation in which the system is in a deadlocked state yet has no way of recognizing what has happened. In this case, the undetected deadlock will cause the system's performance to deteriorate, because resources are being held by threads that cannot run and because more and more threads, as they make requests for resources, will enter a deadlocked state. Eventually, the system will stop functioning and will need to be restarted manually.

Although this method may not seem to be a viable approach to the deadlock problem, it is nevertheless used in most operating systems, as mentioned earlier. Expense is one important consideration. Ignoring the possibility of deadlocks is cheaper than the other approaches. Since in many systems, deadlocks occur infrequently (say, once per month), the extra expense of the other methods may not seem worthwhile.

In addition, methods used to recover from other liveness conditions, such as livelock, may be used to recover from deadlock. In some circumstances, a system is suffering from a liveness failure but is not in a deadlocked state. We see this situation, for example, with a real-time thread running at the highest priority (or any thread running on a nonpreemptive scheduler) and never returning control to the operating system. The system must have manual recovery methods for such conditions and may simply use those techniques for deadlock recovery.

## 8.5 Deadlock Prevention

As we noted in Section 8.3.1, for a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can *prevent* the occurrence of a deadlock. We elaborate on this approach by examining each of the four necessary conditions separately.

### 8.5.1 Mutual Exclusion

The mutual-exclusion condition must hold. That is, at least one resource must be nonsharable. Sharable resources do not require mutually exclusive access and thus cannot be involved in a deadlock. Read-only files are a good example of a sharable resource. If several threads attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. A thread never needs to wait for a sharable resource. In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically nonsharable. For example, a mutex lock cannot be simultaneously shared by several threads.

### 8.5.2 Hold and Wait

To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a thread requests a resource, it does not hold any other resources. One protocol that we can use requires each thread to request and be allocated all its resources before it begins execution. This is, of course, impractical for most applications due to the dynamic nature of requesting resources.

An alternative protocol allows a thread to request resources only when it has none. A thread may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated.

Both these protocols have two main disadvantages. First, resource utilization may be low, since resources may be allocated but unused for a long period. For example, a thread may be allocated a mutex lock for its entire execution, yet only require it for a short duration. Second, starvation is possible. A thread that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other thread.

### 8.5.3 No Preemption

The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated. To ensure that this condition does not hold, we can use the following protocol. If a thread is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the thread must wait), then all resources the thread is currently holding are preempted. In other words, these resources are implicitly released. The preempted resources are added to the list of resources for which the thread is waiting. The thread will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

Alternatively, if a thread requests some resources, we first check whether they are available. If they are, we allocate them. If they are not, we check whether they are allocated to some other thread that is waiting for additional resources. If so, we preempt the desired resources from the waiting thread and allocate them to the requesting thread. If the resources are neither available nor held by a waiting thread, the requesting thread must wait. While it is waiting, some of its resources may be preempted, but only if another thread requests them. A thread can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting.

This protocol is often applied to resources whose state can be easily saved and restored later, such as CPU registers and database transactions. It cannot generally be applied to such resources as mutex locks and semaphores, precisely the type of resources where deadlock occurs most commonly.

### 8.5.4 Circular Wait

The three options presented thus far for deadlock prevention are generally impractical in most situations. However, the fourth and final condition for deadlocks — the circular-wait condition — presents an opportunity for a practical solution by invalidating one of the necessary conditions. One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each thread requests resources in an increasing order of enumeration.

To illustrate, we let $R = \{R_1, R_2, ..., R_m\}$ be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering. Formally, we define a one-to-one function $F: R \rightarrow N$, where $N$ is the set of natural numbers. We can accomplish this scheme in an application program by developing an ordering among all synchronization objects in the system. For example, the lock ordering in the Pthread program shown in Figure 8.1 could be

$$F(\texttt{first\_mutex}) = 1$$
$$F(\texttt{second\_mutex}) = 5$$

We can now consider the following protocol to prevent deadlocks: Each thread can request resources only in an increasing order of enumeration. That is, a thread can initially request an instance of a resource—say, $R_i$. After that, the thread can request an instance of resource $R_j$ if and only if $F(R_j) > F(R_i)$. For example, using the function defined above, a thread that wants to use both `first_mutex` and `second_mutex` at the same time must first request `first_mutex` and then `second_mutex`. Alternatively, we can require that a thread requesting an instance of resource $R_j$ must have released any resources $R_i$ such that $F(R_i) \geq F(R_j)$. Note also that if several instances of the same resource type are needed, a ***single*** request for all of them must be issued.

If these two protocols are used, then the circular-wait condition cannot hold. We can demonstrate this fact by assuming that a circular wait exists (proof by contradiction). Let the set of threads involved in the circular wait be $\{T_0, T_1, ..., T_n\}$, where $T_i$ is waiting for a resource $R_i$, which is held by thread $T_{i+1}$. (Modulo arithmetic is used on the indexes, so that $T_n$ is waiting for a resource $R_n$ held by $T_0$.) Then, since thread $T_{i+1}$ is holding resource $R_i$ while requesting resource $R_{i+1}$, we must have $F(R_i) < F(R_{i+1})$ for all $i$. But this condition means that $F(R_0) < F(R_1) < ... < F(R_n) < F(R_0)$. By transitivity, $F(R_0) < F(R_0)$, which is impossible. Therefore, there can be no circular wait.

Keep in mind that developing an ordering, or hierarchy, does not in itself prevent deadlock. It is up to application developers to write programs that follow the ordering. However, establishing a lock ordering can be difficult, especially on a system with hundreds—or thousands—of locks. To address this challenge, many Java developers have adopted the strategy of using the method `System.identityHashCode(Object)` (which returns the default hash code value of the `Object` parameter it has been passed) as the function for ordering lock acquisition.

It is also important to note that imposing a lock ordering does not guarantee deadlock prevention if locks can be acquired dynamically. For example, assume we have a function that transfers funds between two accounts. To prevent a race condition, each account has an associated mutex lock that is obtained from a `get_lock()` function such as that shown in Figure 8.7. Deadlock is possible if two threads simultaneously invoke the `transaction()` function, transposing different accounts. That is, one thread might invoke

```
transaction(checking_account, savings_account, 25.0)
```

and another might invoke

```
transaction(savings_account, checking_account, 50.0)
```

```
void transaction(Account from, Account to, double amount)
{
  mutex lock1, lock2;
  lock1 = get_lock(from);
  lock2 = get_lock(to);

  acquire(lock1);
    acquire(lock2);

      withdraw(from, amount);
      deposit(to, amount);

    release(lock2);
  release(lock1);
}
```

**Figure 8.7** Deadlock example with lock ordering.

## 8.6 Deadlock Avoidance

Deadlock-prevention algorithms, as discussed in Section 8.5, prevent deadlocks by limiting how requests can be made. The limits ensure that at least one of the necessary conditions for deadlock cannot occur. Possible side effects of preventing deadlocks by this method, however, are low device utilization and reduced system throughput.

An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested. For example, in a system with resources $R1$ and $R2$, the system might need to know that thread $P$ will request first $R1$ and then $R2$ before releasing both resources, whereas thread $Q$ will request $R2$ and then $R1$. With this knowledge of the complete sequence of requests and releases for each thread, the system can decide for each request whether or not the thread should wait in order to avoid a possible future deadlock. Each request requires that in making this decision the system consider the resources currently available, the resources currently allocated to each thread, and the future requests and releases of each thread.

The various algorithms that use this approach differ in the amount and type of information required. The simplest and most useful model requires that each thread declare the ***maximum number*** of resources of each type that it may need. Given this a priori information, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state. A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist. The resource-allocation ***state*** is defined by the number of available and allocated resources and the maximum demands of the threads. In the following sections, we explore two deadlock-avoidance algorithms.

### *LINUX LOCKDEP TOOL*

Although ensuring that resources are acquired in the proper order is the responsibility of kernel and application developers, certain software can be used to verify that locks are acquired in the proper order. To detect possible deadlocks, Linux provides `lockdep`, a tool with rich functionality that can be used to verify locking order in the kernel. `lockdep` is designed to be enabled on a running kernel as it monitors usage patterns of lock acquisitions and releases against a set of rules for acquiring and releasing locks. Two examples follow, but note that `lockdep` provides significantly more functionality than what is described here:

- The order in which locks are acquired is dynamically maintained by the system. If `lockdep` detects locks being acquired out of order, it reports a possible deadlock condition.
- In Linux, spinlocks can be used in interrupt handlers. A possible source of deadlock occurs when the kernel acquires a spinlock that is also used in an interrupt handler. If the interrupt occurs while the lock is being held, the interrupt handler preempts the kernel code currently holding the lock and then spins while attempting to acquire the lock, resulting in deadlock. The general strategy for avoiding this situation is to disable interrupts on the current processor before acquiring a spinlock that is also used in an interrupt handler. If `lockdep` detects that interrupts are enabled while kernel code acquires a lock that is also used in an interrupt handler, it will report a possible deadlock scenario.

`lockdep` was developed to be used as a tool in developing or modifying code in the kernel and not to be used on production systems, as it can significantly slow down a system. Its purpose is to test whether software such as a new device driver or kernel module provides a possible source of deadlock. The designers of `lockdep` have reported that within a few years of its development in 2006, the number of deadlocks from system reports had been reduced by an order of magnitude.âŁž Although `lockdep` was originally designed only for use in the kernel, recent revisions of this tool can now be used for detecting deadlocks in user applications using Pthreads mutex locks. Further details on the `lockdep` tool can be found at https://www.kernel.org/doc/Documentation/locking/lockdep-design.txt.

### 8.6.1 Safe State

A state is *safe* if the system can allocate resources to each thread (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a **safe sequence**. A sequence of threads $<T_1, T_2, ..., T_n>$ is a safe sequence for the current allocation state if, for each $T_i$, the resource requests that $T_i$ can still make can be satisfied by the currently available resources plus the resources held by all $T_j$, with $j < i$. In this situation, if the resources that $T_i$ needs are not immediately available, then $T_i$ can wait until all $T_j$ have finished. When they have finished, $T_i$ can obtain all of its

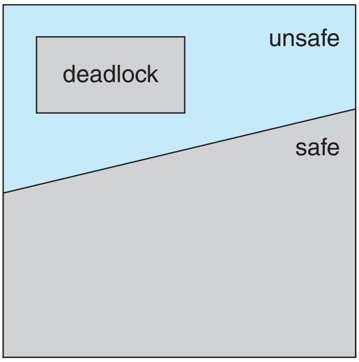


**Figure 8.8** Safe, unsafe, and deadlocked state spaces.

needed resources, complete its designated task, return its allocated resources, and terminate. When $T_i$ terminates, $T_{i+1}$ can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be *unsafe.*

A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks, however (Figure 8.8). An unsafe state ***may*** lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe (and deadlocked) states. In an unsafe state, the operating system cannot prevent threads from requesting resources in such a way that a deadlock occurs. The behavior of the threads controls unsafe states.

To illustrate, consider a system with twelve resources and three threads: $T_0$, $T_1$, and $T_2$. Thread $T_0$ requires ten resources, thread $T_1$ may need as many as four, and thread $T_2$ may need up to nine resources. Suppose that, at time $t_0$, thread $T_0$ is holding five resources, thread $T_1$ is holding two resources, and thread $T_2$ is holding two resources. (Thus, there are three free resources.)

| | Maximum Needs | Current Needs |
|---|---|---|
| $T_0$ | 10 | 5 |
| $T_1$ | 4 | 2 |
| $T_2$ | 9 | 2 |

At time $t_0$, the system is in a safe state. The sequence $<T_1, T_0, T_2>$ satisfies the safety condition. Thread $T_1$ can immediately be allocated all its resources and then return them (the system will then have five available resources); then thread $T_0$ can get all its resources and return them (the system will then have ten available resources); and finally thread $T_2$ can get all its resources and return them (the system will then have all twelve resources available).

A system can go from a safe state to an unsafe state. Suppose that, at time $t_1$, thread $T_2$ requests and is allocated one more resource. The system is no longer in a safe state. At this point, only thread $T_1$ can be allocated all its resources. When it returns them, the system will have only four available resources. Since thread $T_0$ is allocated five resources but has a maximum of ten, it may request five more resources. If it does so, it will have to wait, because they are unavailable. Similarly, thread $T_2$ may request six additional resources and have to wait, resulting in a deadlock. Our mistake was in granting the request from thread $T_2$ for one more resource. If we had made $T_2$ wait until either of the other threads had finished and released its resources, then we could have avoided the deadlock.

Given the concept of a safe state, we can define avoidance algorithms that ensure that the system will never deadlock. The idea is simply to ensure that the system will always remain in a safe state. Initially, the system is in a safe state. Whenever a thread requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or the thread must wait. The request is granted only if the allocation leaves the system in a safe state.

In this scheme, if a thread requests a resource that is currently available, it may still have to wait. Thus, resource utilization may be lower than it would otherwise be.

### 8.6.2 Resource-Allocation-Graph Algorithm

If we have a resource-allocation system with only one instance of each resource type, we can use a variant of the resource-allocation graph defined in Section 8.3.2 for deadlock avoidance. In addition to the request and assignment edges already described, we introduce a new type of edge, called a **claim edge**. A claim edge $T_i \rightarrow R_j$ indicates that thread $T_i$ may request resource $R_j$ at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a dashed line. When thread $T_i$ requests resource $R_j$, the claim edge $T_i \rightarrow R_j$ is converted to a request edge. Similarly, when a resource $R_j$ is released by $T_i$, the assignment edge $R_j \rightarrow T_i$ is reconverted to a claim edge $T_i \rightarrow R_j$.

Note that the resources must be claimed a priori in the system. That is, before thread $T_i$ starts executing, all its claim edges must already appear in the resource-allocation graph. We can relax this condition by allowing a claim edge $T_i \rightarrow R_j$ to be added to the graph only if all the edges associated with thread $T_i$ are claim edges.

Now suppose that thread $T_i$ requests resource $R_j$. The request can be granted only if converting the request edge $T_i \rightarrow R_j$ to an assignment edge $R_j \rightarrow T_i$ does not result in the formation of a cycle in the resource-allocation graph. We check for safety by using a cycle-detection algorithm. An algorithm for detecting a cycle in this graph requires an order of $n^2$ operations, where $n$ is the number of threads in the system.

If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state. In that case, thread $T_i$ will have to wait for its requests to be satisfied.

To illustrate this algorithm, we consider the resource-allocation graph of Figure 8.9. Suppose that $T_2$ requests $R_2$. Although $R_2$ is currently free, we cannot allocate it to $T_2$, since this action will create a cycle in the graph (Figure 8.10). A cycle, as mentioned, indicates that the system is in an unsafe state. If $T_1$ requests $R_2$, and $T_2$ requests $R_1$, then a deadlock will occur.

### 8.6.3 Banker's Algorithm

The resource-allocation-graph algorithm is not applicable to a resource-allocation system with multiple instances of each resource type. The

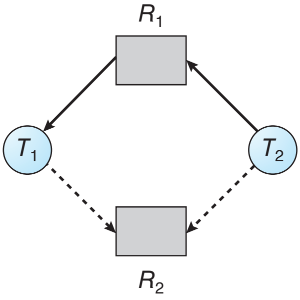


**Figure 8.9** Resource-allocation graph for deadlock avoidance.

deadlock-avoidance algorithm that we describe next is applicable to such a system but is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the **banker's algorithm**. The name was chosen because the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.

When a new thread enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the thread must wait until some other thread releases enough resources.

Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. We need the following data structures, where $n$ is the number of threads in the system and $m$ is the number of resource types:

- **Available**. A vector of length $m$ indicates the number of available resources of each type. If ***Available***[$j$] equals $k$, then $k$ instances of resource type $R_j$ are available.

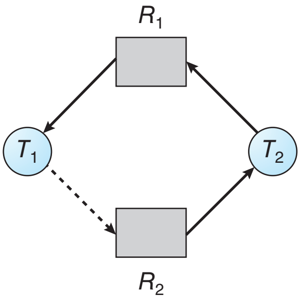


**Figure 8.10** An unsafe state in a resource-allocation graph.

- **Max**. An $n \times m$ matrix defines the maximum demand of each thread. If ***Max***$[i][j]$ equals $k$, then thread $T_i$ may request at most $k$ instances of resource type $R_j$.
- **Allocation**. An $n \times m$ matrix defines the number of resources of each type currently allocated to each thread. If ***Allocation***$[i][j]$ equals $k$, then thread $T_i$ is currently allocated $k$ instances of resource type $R_j$.
- **Need**. An $n \times m$ matrix indicates the remaining resource need of each thread. If ***Need***$[i][j]$ equals $k$, then thread $T_i$ may need $k$ more instances of resource type $R_j$ to complete its task. Note that ***Need***$[i][j]$ equals ***Max***$[i][j]$ − ***Allocation***$[i][j]$.

These data structures vary over time in both size and value.

To simplify the presentation of the banker's algorithm, we next establish some notation. Let $X$ and $Y$ be vectors of length $n$. We say that $X \leq Y$ if and only if $X[i] \leq Y[i]$ for all $i = 1, 2, ..., n$. For example, if $X = (1,7,3,2)$ and $Y = (0,3,2,1)$, then $Y \leq X$. In addition, $Y < X$ if $Y \leq X$ and $Y \neq X$.

We can treat each row in the matrices ***Allocation*** and ***Need*** as vectors and refer to them as ***Allocation***$_i$ and ***Need***$_i$. The vector ***Allocation***$_i$ specifies the resources currently allocated to thread $T_i$; the vector ***Need***$_i$ specifies the additional resources that thread $T_i$ may still request to complete its task.

### 8.6.3.1 Safety Algorithm

We can now present the algorithm for finding out whether or not a system is in a safe state. This algorithm can be described as follows:

1. Let ***Work*** and ***Finish*** be vectors of length $m$ and $n$, respectively. Initialize ***Work*** = ***Available*** and ***Finish***$[i]$ = ***false*** for $i = 0, 1, ..., n - 1$.
2. Find an index $i$ such that both
   a. ***Finish***$[i]$ == ***false***
   b. ***Need***$_i \leq$ ***Work***

   If no such $i$ exists, go to step 4.
3. ***Work*** = ***Work*** + ***Allocation***$_i$
   ***Finish***$[i]$ = ***true***
   Go to step 2.
4. If ***Finish***$[i]$ == ***true*** for all $i$, then the system is in a safe state.

This algorithm may require an order of $m \times n^2$ operations to determine whether a state is safe.

### 8.6.3.2 Resource-Request Algorithm

Next, we describe the algorithm for determining whether requests can be safely granted. Let ***Request***$_i$ be the request vector for thread $T_i$. If ***Request***$_i$ $[j]$ == $k$, then thread $T_i$ wants $k$ instances of resource type $R_j$. When a request for resources is made by thread $T_i$, the following actions are taken:

1. If $\textbf{\textit{Request}}_i \leq \textbf{\textit{Need}}_i$, go to step 2. Otherwise, raise an error condition, since the thread has exceeded its maximum claim.
2. If $\textbf{\textit{Request}}_i \leq \textbf{\textit{Available}}$, go to step 3. Otherwise, $T_i$ must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to thread $T_i$ by modifying the state as follows:

$$\begin{aligned}&\textbf{\textit{Available}} = \textbf{\textit{Available}} - \textbf{\textit{Request}}_i\\&\textbf{\textit{Allocation}}_i = \textbf{\textit{Allocation}}_i + \textbf{\textit{Request}}_i\\&\textbf{\textit{Need}}_i = \textbf{\textit{Need}}_i - \textbf{\textit{Request}}_i\end{aligned}$$

If the resulting resource-allocation state is safe, the transaction is completed, and thread $T_i$ is allocated its resources. However, if the new state is unsafe, then $T_i$ must wait for $\textbf{\textit{Request}}_i$, and the old resource-allocation state is restored.

#### 8.6.3.3 An Illustrative Example

To illustrate the use of the banker's algorithm, consider a system with five threads $T_0$ through $T_4$ and three resource types $A$, $B$, and $C$. Resource type $A$ has ten instances, resource type $B$ has five instances, and resource type $C$ has seven instances. Suppose that the following snapshot represents the current state of the system:

| | ***Allocation*** | ***Max*** | ***Available*** |
|---|---|---|---|
| | *A B C* | *A B C* | *A B C* |
| $T_0$ | 0 1 0 | 7 5 3 | 3 3 2 |
| $T_1$ | 2 0 0 | 3 2 2 | |
| $T_2$ | 3 0 2 | 9 0 2 | |
| $T_3$ | 2 1 1 | 2 2 2 | |
| $T_4$ | 0 0 2 | 4 3 3 | |

The content of the matrix ***Need*** is defined to be ***Max*** − ***Allocation*** and is as follows:

| | ***Need*** |
|---|---|
| | *A B C* |
| $T_0$ | 7 4 3 |
| $T_1$ | 1 2 2 |
| $T_2$ | 6 0 0 |
| $T_3$ | 0 1 1 |
| $T_4$ | 4 3 1 |

We claim that the system is currently in a safe state. Indeed, the sequence $<T_1, T_3, T_4, T_2, T_0>$ satisfies the safety criteria. Suppose now that thread $T_1$ requests one additional instance of resource type $A$ and two instances of resource type $C$, so $\textbf{\textit{Request}}_1 = (1,0,2)$. To decide whether this request can be immediately granted, we first check that $\textbf{\textit{Request}}_1 \leq \textbf{\textit{Available}}$—that is, that

(1,0,2) ≤ (3,3,2), which is true. We then pretend that this request has been fulfilled, and we arrive at the following new state:

| | *Allocation* | *Need* | *Available* |
|---|---|---|---|
| | *A B C* | *A B C* | *A B C* |
| $T_0$ | 0 1 0 | 7 4 3 | 2 3 0 |
| $T_1$ | 3 0 2 | 0 2 0 | |
| $T_2$ | 3 0 2 | 6 0 0 | |
| $T_3$ | 2 1 1 | 0 1 1 | |
| $T_4$ | 0 0 2 | 4 3 1 | |

We must determine whether this new system state is safe. To do so, we execute our safety algorithm and find that the sequence $<T_1, T_3, T_4, T_0, T_2>$ satisfies the safety requirement. Hence, we can immediately grant the request of thread $T_1$.

You should be able to see, however, that when the system is in this state, a request for (3,3,0) by $T_4$ cannot be granted, since the resources are not available. Furthermore, a request for (0,2,0) by $T_0$ cannot be granted, even though the resources are available, since the resulting state is unsafe.

We leave it as a programming exercise for students to implement the banker's algorithm.

## 8.7 Deadlock Detection

If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm, then a deadlock situation may occur. In this environment, the system may provide:

- An algorithm that examines the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock

Next, we discuss these two requirements as they pertain to systems with only a single instance of each resource type, as well as to systems with several instances of each resource type. At this point, however, we note that a detection-and-recovery scheme requires overhead that includes not only the run-time costs of maintaining the necessary information and executing the detection algorithm but also the potential losses inherent in recovering from a deadlock.

### 8.7.1 Single Instance of Each Resource Type

If all resources have only a single instance, then we can define a deadlock-detection algorithm that uses a variant of the resource-allocation graph, called a **wait-for** graph. We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.

More precisely, an edge from $T_i$ to $T_j$ in a wait-for graph implies that thread $T_i$ is waiting for thread $T_j$ to release a resource that $T_i$ needs. An edge $T_i \rightarrow T_j$

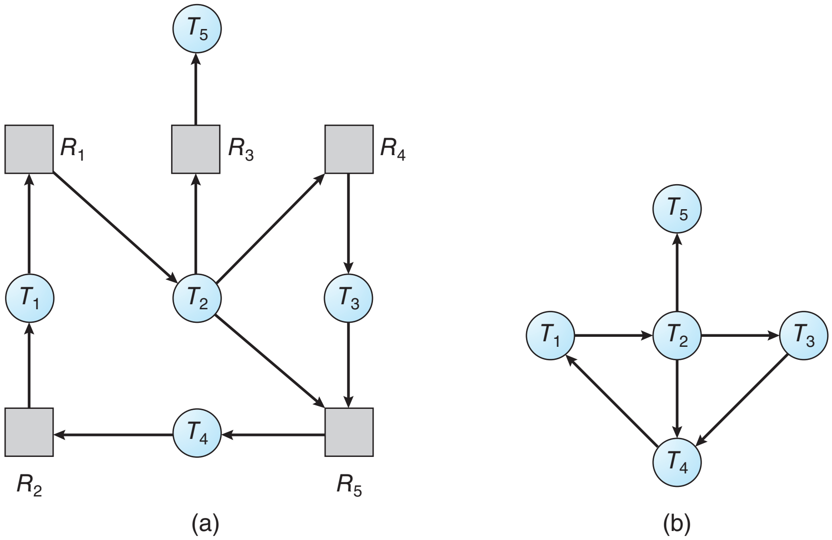


**Figure 8.11** (a) Resource-allocation graph. (b) Corresponding wait-for graph.

exists in a wait-for graph if and only if the corresponding resource-allocation graph contains two edges $T_i \rightarrow R_q$ and $R_q \rightarrow T_j$ for some resource $R_q$. In Figure 8.11, we present a resource-allocation graph and the corresponding wait-for graph.

As before, a deadlock exists in the system if and only if the wait-for graph contains a cycle. To detect deadlocks, the system needs to ***maintain*** the wait-for graph and periodically ***invoke an algorithm*** that searches for a cycle in the graph. An algorithm to detect a cycle in a graph requires $O(n^2)$ operations, where $n$ is the number of vertices in the graph.

The BCC toolkit described in Section 2.10.4 provides a tool that can detect potential deadlocks with Pthreads mutex locks in a user process running on a Linux system. The BCC tool `deadlock_detector` operates by inserting probes which trace calls to the `pthread_mutex_lock()` and `pthread_mutex_unlock()` functions. When the specified process makes a call to either function, `deadlock_detector` constructs a wait-for graph of mutex locks in that process, and reports the possibility of deadlock if it detects a cycle in the graph.

### 8.7.2 Several Instances of a Resource Type

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type. We turn now to a deadlock-detection algorithm that is applicable to such a system. The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm (Section 8.6.3):

- **Available**. A vector of length $m$ indicates the number of available resources of each type.

### *DEADLOCK DETECTION USING JAVA THREAD DUMPS*

Although Java does not provide explicit support for deadlock detection, a **thread dump** can be used to analyze a running program to determine if there is a deadlock. A thread dump is a useful debugging tool that displays a snapshot of the states of all threads in a Java application. Java thread dumps also show locking information, including which locks a blocked thread is waiting to acquire. When a thread dump is generated, the JVM searches the wait-for graph to detect cycles, reporting any deadlocks it detects. To generate a thread dump of a running application, from the command line enter:

`Ctrl-L` (UNIX, Linux, or macOS)
`Ctrl-Break` (Windows)

In the source-code download for this text, we provide a Java example of the program shown in Figure 8.1 and describe how to generate a thread dump that reports the deadlocked Java threads.

- **Allocation**. An $n \times m$ matrix defines the number of resources of each type currently allocated to each thread.
- **Request**. An $n \times m$ matrix indicates the current request of each thread. If $\boldsymbol{Request}[i][j]$ equals $k$, then thread $T_i$ is requesting $k$ more instances of resource type $R_j$.

The $\leq$ relation between two vectors is defined as in Section 8.6.3. To simplify notation, we again treat the rows in the matrices ***Allocation*** and ***Request*** as vectors; we refer to them as $\boldsymbol{Allocation}_i$ and $\boldsymbol{Request}_i$. The detection algorithm described here simply investigates every possible allocation sequence for the threads that remain to be completed. Compare this algorithm with the banker's algorithm of Section 8.6.3.

1. Let ***Work*** and ***Finish*** be vectors of length $m$ and $n$, respectively. Initialize $\boldsymbol{Work} = \boldsymbol{Available}$. For $i = 0, 1, ..., n-1$, if $\boldsymbol{Allocation}_i \neq 0$, then $\boldsymbol{Finish}[i] = \boldsymbol{false}$. Otherwise, $\boldsymbol{Finish}[i] = \boldsymbol{true}$.
2. Find an index $i$ such that both
   a. $\boldsymbol{Finish}[i] == \boldsymbol{false}$
   b. $\boldsymbol{Request}_i \leq \boldsymbol{Work}$

   If no such $i$ exists, go to step 4.
3. $\boldsymbol{Work} = \boldsymbol{Work} + \boldsymbol{Allocation}_i$
   $\boldsymbol{Finish}[i] = \boldsymbol{true}$
   Go to step 2.
4. If $\boldsymbol{Finish}[i] == \boldsymbol{false}$ for some $i$, $0 \leq i < n$, then the system is in a deadlocked state. Moreover, if $\boldsymbol{Finish}[i] == \boldsymbol{false}$, then thread $T_i$ is deadlocked.

This algorithm requires an order of $m \times n^2$ operations to detect whether the system is in a deadlocked state.

You may wonder why we reclaim the resources of thread $T_i$ (in step 3) as soon as we determine that $\textbf{\textit{Request}}_i \leq \textbf{\textit{Work}}$ (in step 2b). We know that $T_i$ is currently ***not*** involved in a deadlock (since $\textbf{\textit{Request}}_i \leq \textbf{\textit{Work}}$). Thus, we take an optimistic attitude and assume that $T_i$ will require no more resources to complete its task; it will thus soon return all currently allocated resources to the system. If our assumption is incorrect, a deadlock may occur later. That deadlock will be detected the next time the deadlock-detection algorithm is invoked.

To illustrate this algorithm, we consider a system with five threads $T_0$ through $T_4$ and three resource types $A$, $B$, and $C$. Resource type $A$ has seven instances, resource type $B$ has two instances, and resource type $C$ has six instances. The following snapshot represents the current state of the system:

| | ***Allocation*** | ***Request*** | ***Available*** |
|---|---|---|---|
| | *A B C* | *A B C* | *A B C* |
| $T_0$ | 0 1 0 | 0 0 0 | 0 0 0 |
| $T_1$ | 2 0 0 | 2 0 2 | |
| $T_2$ | 3 0 3 | 0 0 0 | |
| $T_3$ | 2 1 1 | 1 0 0 | |
| $T_4$ | 0 0 2 | 0 0 2 | |

We claim that the system is not in a deadlocked state. Indeed, if we execute our algorithm, we will find that the sequence $<T_0, T_2, T_3, T_1, T_4>$ results in ***Finish***[$i$] == *true* for all $i$.

Suppose now that thread $T_2$ makes one additional request for an instance of type $C$. The ***Request*** matrix is modified as follows:

| | ***Request*** |
|---|---|
| | *A B C* |
| $T_0$ | 0 0 0 |
| $T_1$ | 2 0 2 |
| $T_2$ | 0 0 1 |
| $T_3$ | 1 0 0 |
| $T_4$ | 0 0 2 |

We claim that the system is now deadlocked. Although we can reclaim the resources held by thread $T_0$, the number of available resources is not sufficient to fulfill the requests of the other threads. Thus, a deadlock exists, consisting of threads $T_1$, $T_2$, $T_3$, and $T_4$.

### 8.7.3 Detection-Algorithm Usage

When should we invoke the detection algorithm? The answer depends on two factors:

1. How ***often*** is a deadlock likely to occur?
2. How ***many*** threads will be affected by deadlock when it happens?

### *MANAGING DEADLOCK IN DATABASES*

Database systems provide a useful illustration of how both open-source and commercial software manage deadlock. Updates to a database may be performed as ***transactions***, and to ensure data integrity, locks are typically used. A transaction may involve several locks, so it comes as no surprise that deadlocks are possible in a database with multiple concurrent transactions. To manage deadlock, most transactional database systems include a deadlock detection and recovery mechanism. The database server will periodically search for cycles in the wait-for graph to detect deadlock among a set of transactions. When deadlock is detected, a victim is selected and the transaction is aborted and rolled back, releasing the locks held by the victim transaction and freeing the remaining transactions from deadlock. Once the remaining transactions have resumed, the aborted transaction is reissued. Choice of a victim transaction depends on the database system; for instance, MySQL attempts to select transactions that minimize the number of rows being inserted, updated, or deleted.

If deadlocks occur frequently, then the detection algorithm should be invoked frequently. Resources allocated to deadlocked threads will be idle until the deadlock can be broken. In addition, the number of threads involved in the deadlock cycle may grow.

Deadlocks occur only when some thread makes a request that cannot be granted immediately. This request may be the final request that completes a chain of waiting threads. In the extreme, then, we can invoke the deadlock-detection algorithm every time a request for allocation cannot be granted immediately. In this case, we can identify not only the deadlocked set of threads but also the specific thread that "caused" the deadlock. (In reality, each of the deadlocked threads is a link in the cycle in the resource graph, so all of them, jointly, caused the deadlock.) If there are many different resource types, one request may create many cycles in the resource graph, each cycle completed by the most recent request and "caused" by the one identifiable thread.

Of course, invoking the deadlock-detection algorithm for every resource request will incur considerable overhead in computation time. A less expensive alternative is simply to invoke the algorithm at defined intervals—for example, once per hour or whenever CPU utilization drops below 40 percent. (A deadlock eventually cripples system throughput and causes CPU utilization to drop.) If the detection algorithm is invoked at arbitrary points in time, the resource graph may contain many cycles. In this case, we generally cannot tell which of the many deadlocked threads "caused" the deadlock.

## 8.8 Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, several alternatives are available. One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually. Another possibility is to let the system **recover** from the deadlock automatically. There are two options for breaking a deadlock. One is simply to abort one or more threads to break the circular wait. The other is to preempt some resources from one or more of the deadlocked threads.

### 8.8.1 Process and Thread Termination

To eliminate deadlocks by aborting a process or thread, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

- **Abort all deadlocked processes**. This method clearly will break the deadlock cycle, but at great expense. The deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.
- **Abort one process at a time until the deadlock cycle is eliminated**. This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it may leave that file in an incorrect state. Similarly, if the process was in the midst of updating shared data while holding a mutex lock, the system must restore the status of the lock as being available, although no guarantees can be made regarding the integrity of the shared data.

If the partial termination method is used, then we must determine which deadlocked process (or processes) should be terminated. This determination is a policy decision, similar to CPU-scheduling decisions. The question is basically an economic one; we should abort those processes whose termination will incur the minimum cost. Unfortunately, the term ***minimum cost*** is not a precise one. Many factors may affect which process is chosen, including:

1. What the priority of the process is
2. How long the process has computed and how much longer the process will compute before completing its designated task
3. How many and what types of resources the process has used (for example, whether the resources are simple to preempt)
4. How many more resources the process needs in order to complete
5. How many processes will need to be terminated

### 8.8.2 Resource Preemption

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

If preemption is required to deal with deadlocks, then three issues need to be addressed:

1. **Selecting a victim**. Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed.

2. **Rollback**. If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state.

   Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: abort the process and then restart it. Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes.

3. **Starvation**. How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?

   In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a starvation situation any practical system must address. Clearly, we must ensure that a process can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

## 8.9 Summary

- Deadlock occurs in a set of processes when every process in the set is waiting for an event that can only be caused by another process in the set.
- There are four necessary conditions for deadlock: (1) mutual exclusion, (2) hold and wait, (3) no preemption, and (4) circular wait. Deadlock is only possible when all four conditions are present.
- Deadlocks can be modeled with resource-allocation graphs, where a cycle indicates deadlock.
- Deadlocks can be prevented by ensuring that one of the four necessary conditions for deadlock cannot occur. Of the four necessary conditions, eliminating the circular wait is the only practical approach.
- Deadlock can be avoided by using the banker's algorithm, which does not grant resources if doing so would lead the system into an unsafe state where deadlock would be possible.
- A deadlock-detection algorithm can evaluate processes and resources on a running system to determine if a set of processes is in a deadlocked state.
- If deadlock does occur, a system can attempt to recover from the deadlock by either aborting one of the processes in the circular wait or preempting resources that have been assigned to a deadlocked process.

## Practice Exercises

**8.1** List three examples of deadlocks that are not related to a computer-system environment.

**8.2** Suppose that a system is in an unsafe state. Show that it is possible for the threads to complete their execution without entering a deadlocked state.

**8.3** Consider the following snapshot of a system:

| | *Allocation* | *Max* | *Available* |
|---|---|---|---|
| | *A B C D* | *A B C D* | *A B C D* |
| $T_0$ | 0 0 1 2 | 0 0 1 2 | 1 5 2 0 |
| $T_1$ | 1 0 0 0 | 1 7 5 0 | |
| $T_2$ | 1 3 5 4 | 2 3 5 6 | |
| $T_3$ | 0 6 3 2 | 0 6 5 2 | |
| $T_4$ | 0 0 1 4 | 0 6 5 6 | |

Answer the following questions using the banker's algorithm:

a. What is the content of the matrix ***Need***?

b. Is the system in a safe state?

c. If a request from thread $T_1$ arrives for (0,4,2,0), can the request be granted immediately?

**8.4** A possible method for preventing deadlocks is to have a single, higher-order resource that must be requested before any other resource. For example, if multiple threads attempt to access the synchronization objects $A \cdots E$, deadlock is possible. (Such synchronization objects may include mutexes, semaphores, condition variables, and the like.) We can prevent deadlock by adding a sixth object $F$. Whenever a thread wants to acquire the synchronization lock for any object $A \cdots E$, it must first acquire the lock for object $F$. This solution is known as **containment**: the locks for objects $A \cdots E$ are contained within the lock for object $F$. Compare this scheme with the circular-wait scheme of Section 8.5.4.

**8.5** Prove that the safety algorithm presented in Section 8.6.3 requires an order of $m \times n^2$ operations.

**8.6** Consider a computer system that runs 5,000 jobs per month and has no deadlock-prevention or deadlock-avoidance scheme. Deadlocks occur about twice per month, and the operator must terminate and rerun about ten jobs per deadlock. Each job is worth about two dollars (in CPU time), and the jobs terminated tend to be about half done when they are aborted.

A systems programmer has estimated that a deadlock-avoidance algorithm (like the banker's algorithm) could be installed in the system with an increase of about 10 percent in the average execution time per job. Since the machine currently has 30 percent idle time, all 5,000 jobs per month could still be run, although turnaround time would increase by about 20 percent on average.

a. What are the arguments for installing the deadlock-avoidance algorithm?

b. What are the arguments against installing the deadlock-avoidance algorithm?

**8.7** Can a system detect that some of its threads are starving? If you answer "yes," explain how it can. If you answer "no," explain how the system can deal with the starvation problem.

**8.8** Consider the following resource-allocation policy. Requests for and releases of resources are allowed at any time. If a request for resources cannot be satisfied because the resources are not available, then we check any threads that are blocked waiting for resources. If a blocked thread has the desired resources, then these resources are taken away from it and are given to the requesting thread. The vector of resources for which the blocked thread is waiting is increased to include the resources that were taken away.

For example, a system has three resource types, and the vector ***Available*** is initialized to (4,2,2). If thread $T_0$ asks for (2,2,1), it gets them. If $T_1$ asks for (1,0,1), it gets them. Then, if $T_0$ asks for (0,0,1), it is blocked (resource not available). If $T_2$ now asks for (2,0,0), it gets the available one (1,0,0), as well as one that was allocated to $T_0$ (since $T_0$ is blocked). $T_0$'s ***Allocation*** vector goes down to (1,2,1), and its ***Need*** vector goes up to (1,0,1).

a. Can deadlock occur? If you answer "yes," give an example. If you answer "no," specify which necessary condition cannot occur.

b. Can indefinite blocking occur? Explain your answer.

**8.9** Consider the following snapshot of a system:

| | *Allocation* | *Max* |
|---|---|---|
| | *A B C D* | *A B C D* |
| $T_0$ | 3 0 1 4 | 5 1 1 7 |
| $T_1$ | 2 2 1 0 | 3 2 1 1 |
| $T_2$ | 3 1 2 1 | 3 3 2 1 |
| $T_3$ | 0 5 1 0 | 4 6 1 2 |
| $T_4$ | 4 2 1 2 | 6 3 2 5 |

Using the banker's algorithm, determine whether or not each of the following states is unsafe. If the state is safe, illustrate the order in which the threads may complete. Otherwise, illustrate why the state is unsafe.

a. $\textit{Available} = (0, 3, 0, 1)$

b. $\textit{Available} = (1, 0, 0, 2)$

**8.10** Suppose that you have coded the deadlock-avoidance safety algorithm that determines if a system is in a safe state or not, and now have been asked to implement the deadlock-detection algorithm. Can you do so by simply using the safety algorithm code and redefining $Max_i$ = $Waiting_i$ + $Allocation_i$, where $Waiting_i$ is a vector specifying the resources for which thread *i* is waiting and $Allocation_i$ is as defined in Section 8.6? Explain your answer.

**8.11** Is it possible to have a deadlock involving only one single-threaded process? Explain your answer.

## Further Reading

Most research involving deadlock was conducted many years ago. [Dijkstra (1965)] was one of the first and most influential contributors in the deadlock area.

Details of how the MySQL database manages deadlock can be found at http://dev.mysql.com/.

Details on the `lockdep` tool can be found at https://www.kernel.org/doc/Documentation/locking/lockdep-design.txt.

## Bibliography

**[Dijkstra (1965)]** E. W. Dijkstra, “Cooperating Sequential Processes”, Technical report, Technological University, Eindhoven, the Netherlands (1965).

## Chapter 8 Exercises

**8.12** Consider the traffic deadlock depicted in Figure 8.12.

a. Show that the four necessary conditions for deadlock hold in this example.

b. State a simple rule for avoiding deadlocks in this system.

**8.13** Draw the resource-allocation graph that illustrates deadlock from the program example shown in Figure 8.1 in Section 8.2.

**8.14** In Section 6.8.1, we described a potential deadlock scenario involving processes $P_0$ and $P_1$ and semaphores `S` and `Q`. Draw the resource-allocation graph that illustrates deadlock under the scenario presented in that section.

**8.15** Assume that a multithreaded application uses only reader–writer locks for synchronization. Applying the four necessary conditions for deadlock, is deadlock still possible if multiple reader–writer locks are used?

**8.16** The program example shown in Figure 8.1 doesn't always lead to deadlock. Describe what role the CPU scheduler plays and how it can contribute to deadlock in this program.

**8.17** In Section 8.5.4, we described a situation in which we prevent deadlock by ensuring that all locks are acquired in a certain order. However, we also point out that deadlock is possible in this situation if two threads simultaneously invoke the `transaction()` function. Fix the `transaction()` function to prevent deadlocks.

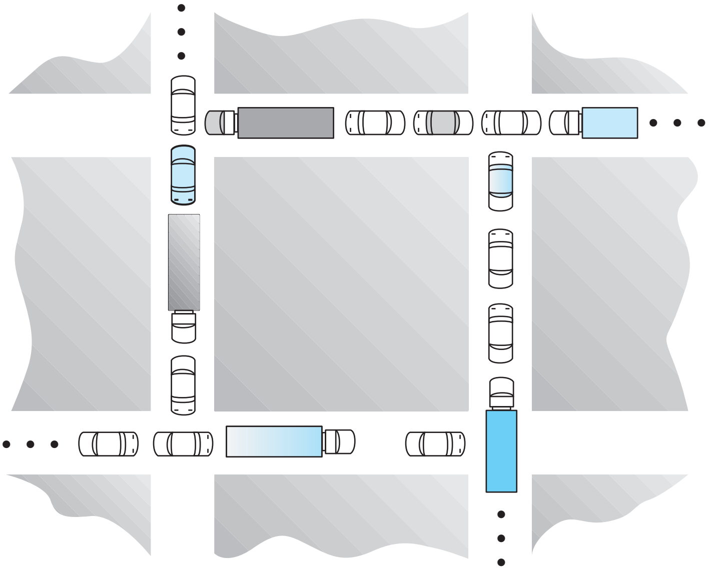


**Figure 8.11** Traffic deadlock for Exercise 8.12.

**8.18** Which of the six resource-allocation graphs shown in Figure 8.12 illustrate deadlock? For those situations that are deadlocked, provide the cycle of threads and resources. Where there is not a deadlock situation, illustrate the order in which the threads may complete execution.

**8.19** Compare the circular-wait scheme with the various deadlock-avoidance schemes (like the banker's algorithm) with respect to the following issues:

a. Runtime overhead

b. System throughput

**8.20** In a real computer system, neither the resources available nor the demands of threads for resources are consistent over long periods (months). Resources break or are replaced, new processes and threads come and go, and new resources are bought and added to the system. If deadlock is controlled by the banker's algorithm, which of the

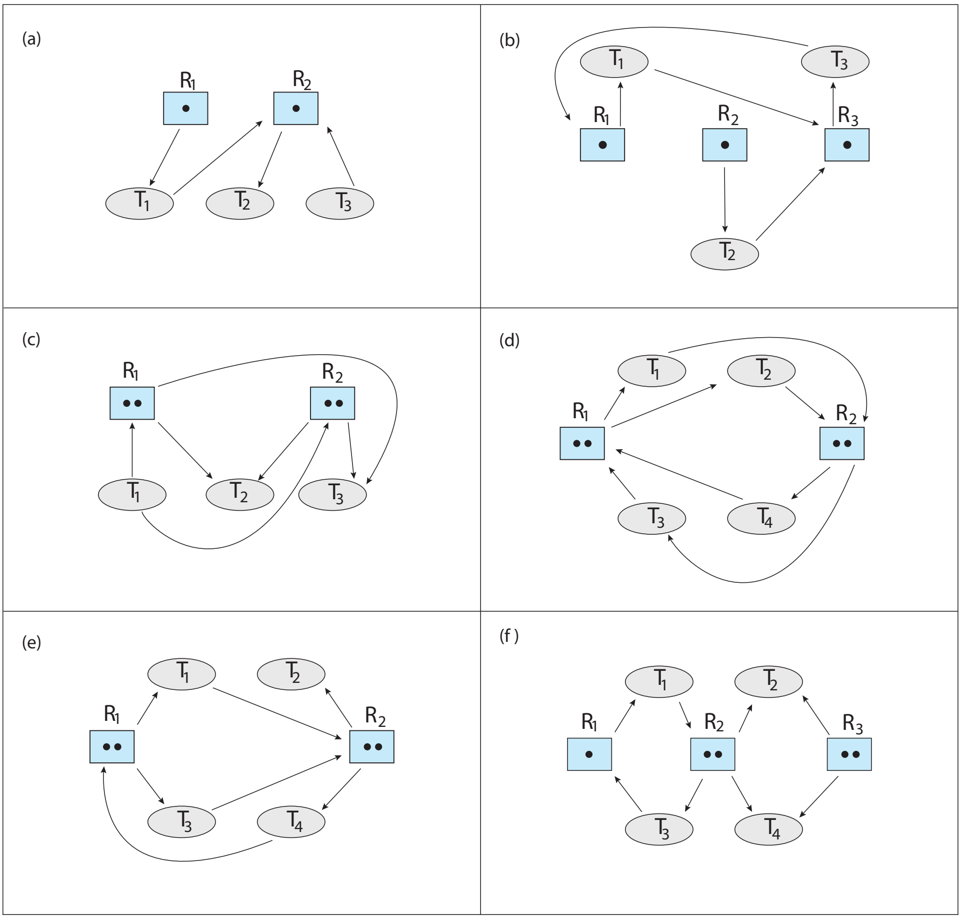


**Figure 8.12** Resource-allocation graphs for Exercise 8.18.

following changes can be made safely (without introducing the possibility of deadlock), and under what circumstances?

a. Increase ***Available*** (new resources added).

b. Decrease ***Available*** (resource permanently removed from system).

c. Increase ***Max*** for one thread (the thread needs or wants more resources than allowed).

d. Decrease ***Max*** for one thread (the thread decides it does not need that many resources).

e. Increase the number of threads.

f. Decrease the number of threads.

**8.21** Consider the following snapshot of a system:

| | ***Allocation*** | ***Max*** |
|---|---|---|
| | *A B C D* | *A B C D* |
| $T_0$ | 2 1 0 6 | 6 3 2 7 |
| $T_1$ | 3 3 1 3 | 5 4 1 5 |
| $T_2$ | 2 3 1 2 | 6 6 1 4 |
| $T_3$ | 1 2 3 4 | 4 3 4 5 |
| $T_4$ | 3 0 3 0 | 7 2 6 1 |

What are the contents of the ***Need*** matrix?

**8.22** Consider a system consisting of four resources of the same type that are shared by three threads, each of which needs at most two resources. Show that the system is deadlock free.

**8.23** Consider a system consisting of $m$ resources of the same type being shared by $n$ threads. A thread can request or release only one resource at a time. Show that the system is deadlock free if the following two conditions hold:

a. The maximum need of each thread is between one resource and $m$ resources.

b. The sum of all maximum needs is less than $m + n$.

**8.24** Consider the version of the dining-philosophers problem in which the chopsticks are placed at the center of the table and any two of them can be used by a philosopher. Assume that requests for chopsticks are made one at a time. Describe a simple rule for determining whether a particular request can be satisfied without causing deadlock given the current allocation of chopsticks to philosophers.

**8.25** Consider again the setting in the preceding exercise. Assume now that each philosopher requires three chopsticks to eat. Resource requests are still issued one at a time. Describe some simple rules for determining whether a particular request can be satisfied without causing deadlock given the current allocation of chopsticks to philosophers.

**8.26** We can obtain the banker's algorithm for a single resource type from the general banker's algorithm simply by reducing the dimensionality of the various arrays by 1.

Show through an example that we cannot implement the multiple-resource-type banker's scheme by applying the single-resource-type scheme to each resource type individually.

**8.27** Consider the following snapshot of a system:

| | *Allocation* | *Max* |
|---|---|---|
| | *A B C D* | *A B C D* |
| $T_0$ | 1 2 0 2 | 4 3 1 6 |
| $T_1$ | 0 1 1 2 | 2 4 2 4 |
| $T_2$ | 1 2 4 0 | 3 6 5 1 |
| $T_3$ | 1 2 0 1 | 2 6 2 3 |
| $T_4$ | 1 0 0 1 | 3 1 1 2 |

Using the banker's algorithm, determine whether or not each of the following states is unsafe. If the state is safe, illustrate the order in which the threads may complete. Otherwise, illustrate why the state is unsafe.

a. *Available* $= (2, 2, 2, 3)$

b. *Available* $= (4, 4, 1, 1)$

c. *Available* $= (3, 0, 1, 4)$

d. *Available* $= (1, 5, 2, 2)$

**8.28** Consider the following snapshot of a system:

| | *Allocation* | *Max* | *Available* |
|---|---|---|---|
| | *A B C D* | *A B C D* | *A B C D* |
| $T_0$ | 3 1 4 1 | 6 4 7 3 | 2 2 2 4 |
| $T_1$ | 2 1 0 2 | 4 2 3 2 | |
| $T_2$ | 2 4 1 3 | 2 5 3 3 | |
| $T_3$ | 4 1 1 0 | 6 3 3 2 | |
| $T_4$ | 2 2 2 1 | 5 6 7 5 | |

Answer the following questions using the banker's algorithm:

a. Illustrate that the system is in a safe state by demonstrating an order in which the threads may complete.

b. If a request from thread $T_4$ arrives for $(2, 2, 2, 4)$, can the request be granted immediately?

c. If a request from thread $T_2$ arrives for $(0, 1, 1, 0)$, can the request be granted immediately?

d. If a request from thread $T_3$ arrives for $(2, 2, 1, 2)$, can the request be granted immediately?

**8.29** What is the optimistic assumption made in the deadlock-detection algorithm? How can this assumption be violated?

**8.30** A single-lane bridge connects the two Vermont villages of North Tunbridge and South Tunbridge. Farmers in the two villages use this bridge to deliver their produce to the neighboring town. The bridge can become deadlocked if a northbound and a southbound farmer get on the bridge at the same time. (Vermont farmers are stubborn and are unable to back up.) Using semaphores and/or mutex locks, design an algorithm in pseudocode that prevents deadlock. Initially, do not be concerned about starvation (the situation in which northbound farmers prevent southbound farmers from using the bridge, or vice versa).

**8.31** Modify your solution to Exercise 8.30 so that it is starvation-free.

## Programming Problems

**8.32** Implement your solution to Exercise 8.30 using POSIX synchronization. In particular, represent northbound and southbound farmers as separate threads. Once a farmer is on the bridge, the associated thread will sleep for a random period of time, representing traveling across the bridge. Design your program so that you can create several threads representing the northbound and southbound farmers.

**8.33** In Figure 8.7, we illustrate a `transaction()` function that dynamically acquires locks. In the text, we describe how this function presents difficulties for acquiring locks in a way that avoids deadlock. Using the Java implementation of `transaction()` that is provided in the source-code download for this text, modify it using the `System.identityHashCode()` method so that the locks are acquired in order.

## Programming Projects

### Banker's Algorithm

For this project, you will write a program that implements the banker's algorithm discussed in Section 8.6.3. Customers request and release resources from the bank. The banker will grant a request only if it leaves the system in a safe state. A request that leaves the system in an unsafe state will be denied. Although the code examples that describe this project are illustrated in C, you may also develop a solution using Java.

#### The Banker

The banker will consider requests from $n$ customers for $m$ resources types, as outlined in Section 8.6.3. The banker will keep track of the resources using the following data structures:

```
#define NUMBER_OF_CUSTOMERS 5
#define NUMBER_OF_RESOURCES 4

/* the available amount of each resource */
int available[NUMBER_OF_RESOURCES];

/*the maximum demand of each customer */
int maximum[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES];

/* the amount currently allocated to each customer */
int allocation[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES];

/* the remaining need of each customer */
int need[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES];
```

The banker will grant a request if it satisfies the safety algorithm outlined in Section 8.6.3.1. If a request does not leave the system in a safe state, the banker will deny it. Function prototypes for requesting and releasing resources are as follows:

```
int request_resources(int customer_num, int request[]);

void release_resources(int customer_num, int release[]);
```

The `request_resources()` function should return 0 if successful and −1 if unsuccessful.

## Testing Your Implementation

Design a program that allows the user to interactively enter a request for resources, to release resources, or to output the values of the different data structures (`available`, `maximum`, `allocation`, and `need`) used with the banker's algorithm.

You should invoke your program by passing the number of resources of each type on the command line. For example, if there were four resource types, with ten instances of the first type, five of the second type, seven of the third type, and eight of the fourth type, you would invoke your program as follows:

```
./a.out 10 5 7 8
```

The `available` array would be initialized to these values.

Your program will initially read in a file containing the maximum number of requests for each customer. For example, if there are five customers and four resources, the input file would appear as follows:

```
6,4,7,3
4,2,3,2
2,5,3,3
6,3,3,2
5,6,7,5
```

where each line in the input file represents the maximum request of each resource type for each customer. Your program will initialize the `maximum` array to these values.

Your program will then have the user enter commands responding to a request of resources, a release of resources, or the current values of the different data structures. Use the command '`RQ`' for requesting resources, '`RL`' for releasing resources, and '`*`' to output the values of the different data structures. For example, if customer 0 were to request the resources $(3, 1, 2, 1)$, the following command would be entered:

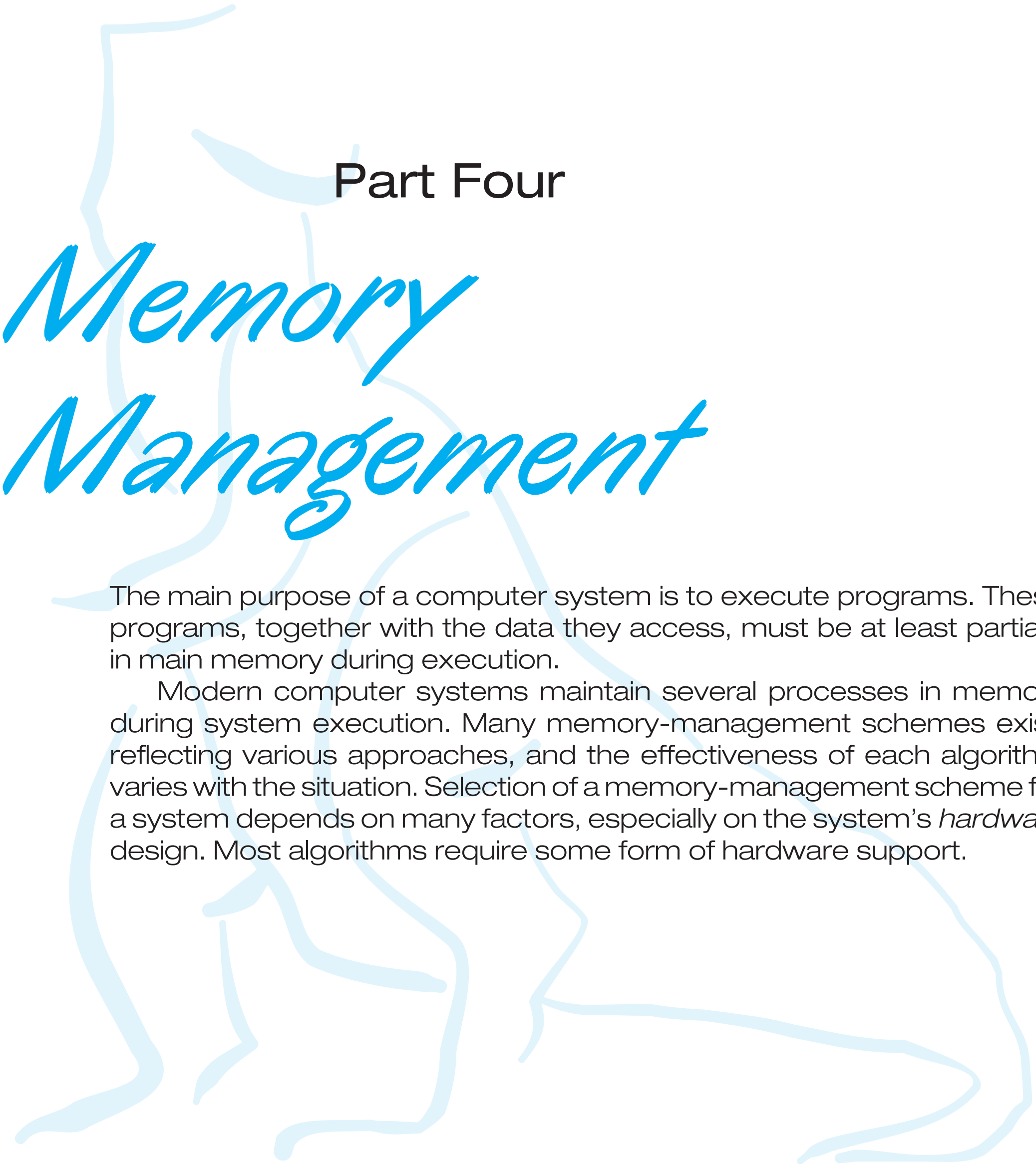
```
RQ 0 3 1 2 1
```

Your program would then output whether the request would be satisfied or denied using the safety algorithm outlined in Section 8.6.3.1.

Similarly, if customer 4 were to release the resources (1, 2, 3, 1), the user would enter the following command:

```
RL 4 1 2 3 1
```

Finally, if the command '`*`' is entered, your program would output the values of the `available`, `maximum`, `allocation`, and `need` arrays.



## Part Four

# Memory Management

The main purpose of a computer system is to execute programs. These programs, together with the data they access, must be at least partially in main memory during execution.

Modern computer systems maintain several processes in memory during system execution. Many memory-management schemes exist, reflecting various approaches, and the effectiveness of each algorithm varies with the situation. Selection of a memory-management scheme for a system depends on many factors, especially on the system's *hardware* design. Most algorithms require some form of hardware support.

CHAPTER 9

# Main Memory

In Chapter 5, we showed how the CPU can be shared by a set of processes. As a result of CPU scheduling, we can improve both the utilization of the CPU and the speed of the computer's response to its users. To realize this increase in performance, however, we must keep many processes in memory—that is, we must share memory.

In this chapter, we discuss various ways to manage memory. The memory-management algorithms vary from a primitive bare-machine approach to a strategy that uses paging. Each approach has its own advantages and disadvantages. Selection of a memory-management method for a specific system depends on many factors, especially on the hardware design of the system. As we shall see, most algorithms require hardware support, leading many systems to have closely integrated hardware and operating-system memory management.

## CHAPTER OBJECTIVES

- Explain the difference between a logical and a physical address and the role of the memory management unit (MMU) in translating addresses.
- Apply first-, best-, and worst-fit strategies for allocating memory contiguously.
- Explain the distinction between internal and external fragmentation.
- Translate logical to physical addresses in a paging system that includes a translation look-aside buffer (TLB).
- Describe hierarchical paging, hashed paging, and inverted page tables.
- Describe address translation for IA-32, x86-64, and ARMv8 architectures.

## 9.1 Background

As we saw in Chapter 1, memory is central to the operation of a modern computer system. Memory consists of a large array of bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter. These instructions may cause additional loading from and storing to specific memory addresses.

A typical instruction-execution cycle, for example, first fetches an instruction from memory. The instruction is then decoded and may cause operands to be fetched from memory. After the instruction has been executed on the operands, results may be stored back in memory. The memory unit sees only a stream of memory addresses; it does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data). Accordingly, we can ignore ***how*** a program generates a memory address. We are interested only in the sequence of memory addresses generated by the running program.

We begin our discussion by covering several issues that are pertinent to managing memory: basic hardware, the binding of symbolic (or virtual) memory addresses to actual physical addresses, and the distinction between logical and physical addresses. We conclude the section with a discussion of dynamic linking and shared libraries.

### 9.1.1 Basic Hardware

Main memory and the registers built into each processing core are the only general-purpose storage that the CPU can access directly. There are machine instructions that take memory addresses as arguments, but none that take disk addresses. Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices. If the data are not in memory, they must be moved there before the CPU can operate on them.

Registers that are built into each CPU core are generally accessible within one cycle of the CPU clock. Some CPU cores can decode instructions and perform simple operations on register contents at the rate of one or more operations per clock tick. The same cannot be said of main memory, which is accessed via a transaction on the memory bus. Completing a memory access may take many cycles of the CPU clock. In such cases, the processor normally needs to **stall**, since it does not have the data required to complete the instruction that it is executing. This situation is intolerable because of the frequency of memory accesses. The remedy is to add fast memory between the CPU and main memory, typically on the CPU chip for fast access. Such a **cache** was described in Section 1.5.5. To manage a cache built into the CPU, the hardware automatically speeds up memory access without any operating-system control. (Recall from Section 5.5.2 that during a memory stall, a multithreaded core can switch from the stalled hardware thread to another hardware thread.)

Not only are we concerned with the relative speed of accessing physical memory, but we also must ensure correct operation. For proper system operation, we must protect the operating system from access by user processes, as well as protect user processes from one another. This protection must be provided by the hardware, because the operating system doesn't usually intervene between the CPU and its memory accesses (because of the resulting performance penalty). Hardware implements this production in several different ways, as we show throughout the chapter. Here, we outline one possible implementation.

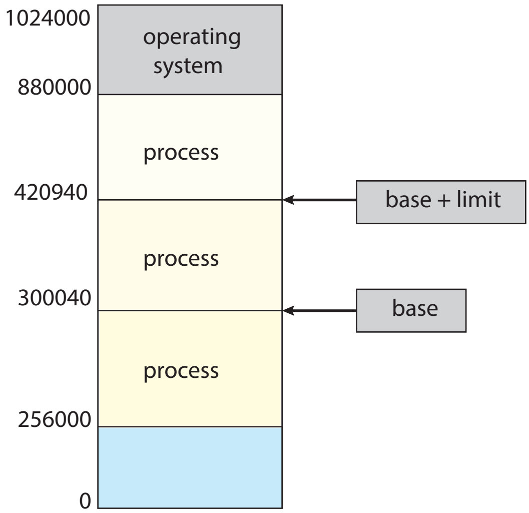


**Figure 9.1** A base and a limit register define a logical address space.

We first need to make sure that each process has a separate memory space. Separate per-process memory space protects the processes from each other and is fundamental to having multiple processes loaded in memory for concurrent execution. To separate memory spaces, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses. We can provide this protection by using two registers, usually a base and a limit, as illustrated in Figure 9.1. The **base register** holds the smallest legal physical memory address; the **limit register** specifies the size of the range. For example, if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive).

Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers. Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error (Figure 9.2). This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.

The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction. Since privileged instructions can be executed only in kernel mode, and since only the operating system executes in kernel mode, only the operating system can load the base and limit registers. This scheme allows the operating system to change the value of the registers but prevents user programs from changing the registers' contents.

The operating system, executing in kernel mode, is given unrestricted access to both operating-system memory and users' memory. This provision allows the operating system to load users' programs into users' memory, to dump out those programs in case of errors, to access and modify parameters of system calls, to perform I/O to and from user memory, and to provide many other services. Consider, for example, that an operating system for a

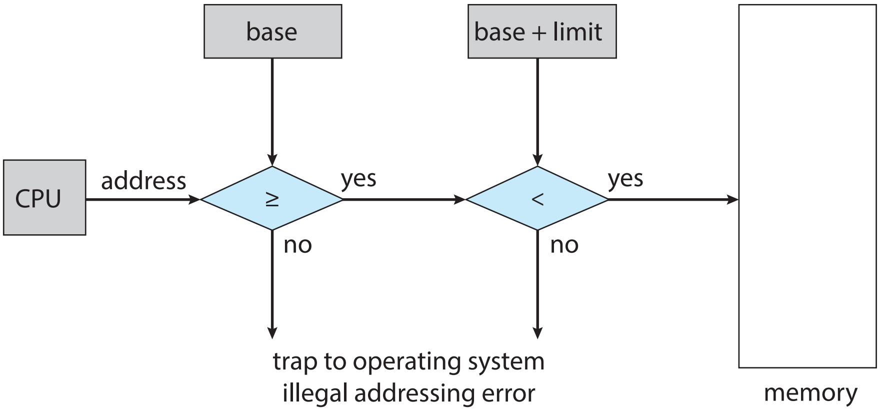


**Figure 9.2** Hardware address protection with base and limit registers.

multiprocessing system must execute context switches, storing the state of one process from the registers into main memory before loading the next process's context from main memory into the registers.

### 9.1.2 Address Binding

Usually, a program resides on a disk as a binary executable file. To run, the program must be brought into memory and placed within the context of a process (as described in Section 2.5), where it becomes eligible for execution on an available CPU. As the process executes, it accesses instructions and data from memory. Eventually, the process terminates, and its memory is reclaimed for use by other processes.

Most systems allow a user process to reside in any part of the physical memory. Thus, although the address space of the computer may start at 00000, the first address of the user process need not be 00000. You will see later how the operating system actually places a process in physical memory.

In most cases, a user program goes through several steps—some of which may be optional—before being executed (Figure 9.3). Addresses may be represented in different ways during these steps. Addresses in the source program are generally symbolic (such as the variable `count`). A compiler typically **binds** these symbolic addresses to relocatable addresses (such as "14 bytes from the beginning of this module"). The linker or loader (see Section 2.5) in turn binds the relocatable addresses to absolute addresses (such as 74014). Each binding is a mapping from one address space to another.

Classically, the binding of instructions and data to memory addresses can be done at any step along the way:

- **Compile time**. If you know at compile time where the process will reside in memory, then **absolute code** can be generated. For example, if you know that a user process will reside starting at location *R*, then the generated compiler code will start at that location and extend up from there. If, at some later time, the starting location changes, then it will be necessary to recompile this code.

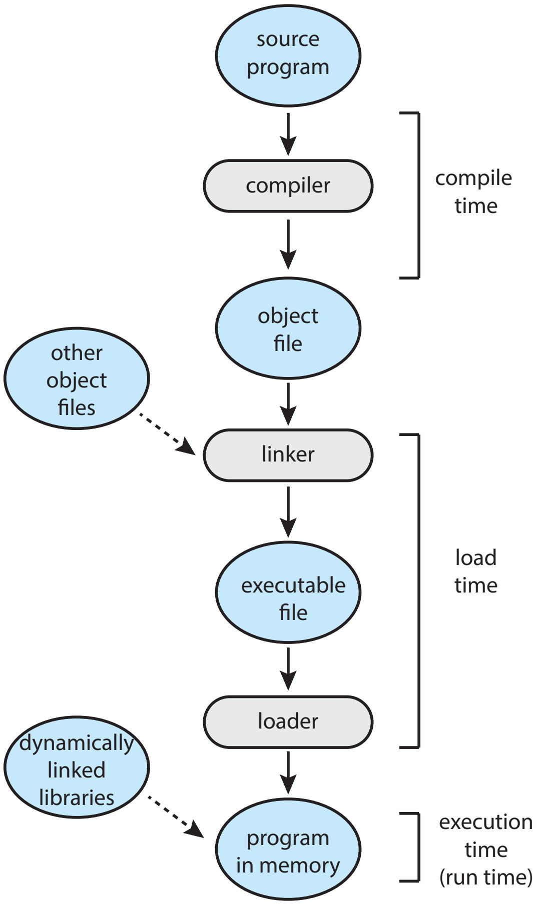

source program
compiler
compile time
object file
other object files
linker
executable file
load time
loader
dynamically linked libraries
program in memory
execution time (run time)

**Figure 9.3** Multistep processing of a user program.

- **Load time**. If it is not known at compile time where the process will reside in memory, then the compiler must generate **relocatable code**. In this case, final binding is delayed until load time. If the starting address changes, we need only reload the user code to incorporate this changed value.
- **Execution time**. If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware must be available for this scheme to work, as will be discussed in Section 9.1.3. Most operating systems use this method.

A major portion of this chapter is devoted to showing how these various bindings can be implemented effectively in a computer system and to discussing appropriate hardware support.

### 9.1.3 Logical Versus Physical Address Space

An address generated by the CPU is commonly referred to as a **logical address**, whereas an address seen by the memory unit—that is, the one loaded into

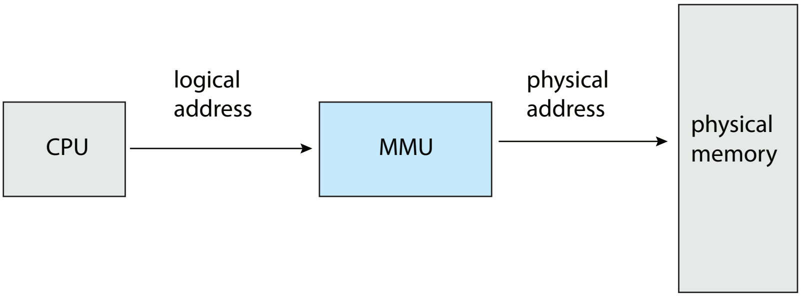


**Figure 9.4** Memory management unit (MMU).

the **memory-address register** of the memory—is commonly referred to as a **physical address**.

Binding addresses at either compile or load time generates identical logical and physical addresses. However, the execution-time address-binding scheme results in differing logical and physical addresses. In this case, we usually refer to the logical address as a **virtual address**. We use ***logical address*** and ***virtual address*** interchangeably in this text. The set of all logical addresses generated by a program is a **logical address space**. The set of all physical addresses corresponding to these logical addresses is a **physical address space**. Thus, in the execution-time address-binding scheme, the logical and physical address spaces differ.

The run-time mapping from virtual to physical addresses is done by a hardware device called the **memory-management unit** (**MMU**) (Figure 9.4). We can choose from many different methods to accomplish such mapping, as we discuss in Section 9.2 through Section 9.3. For the time being, we illustrate this mapping with a simple MMU scheme that is a generalization of the base-register scheme described in Section 9.1.1. The base register is now called a **relocation register**. The value in the relocation register is added to every address generated by a user process at the time the address is sent to memory (see Figure 9.5). For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346.

The user program never accesses the real physical addresses. The program can create a pointer to location 346, store it in memory, manipulate it, and compare it with other addresses—all as the number 346. Only when it is used as a memory address (in an indirect load or store, perhaps) is it relocated relative to the base register. The user program deals with logical addresses. The memory-mapping hardware converts logical addresses into physical addresses. This form of execution-time binding was discussed in Section 9.1.2. The final location of a referenced memory address is not determined until the reference is made.

We now have two different types of addresses: logical addresses (in the range 0 to *max*) and physical addresses (in the range $R + 0$ to $R + max$ for a base value $R$). The user program generates only logical addresses and thinks that the process runs in memory locations from 0 to *max*. However, these logical addresses must be mapped to physical addresses before they are used. The

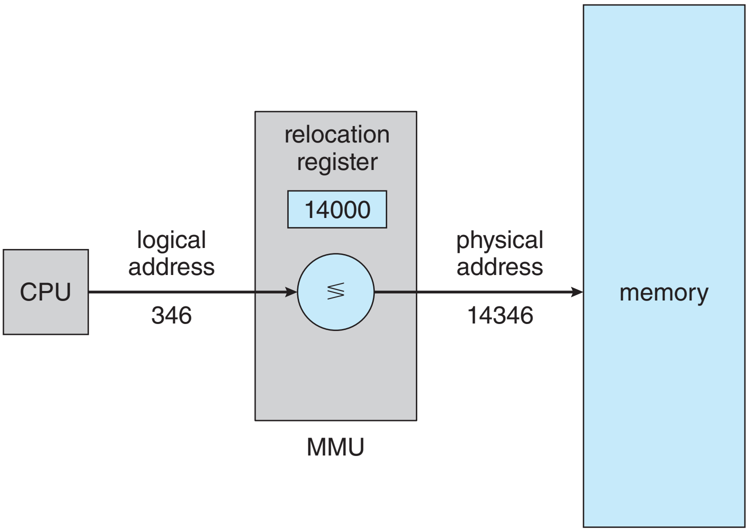


**Figure 9.5** Dynamic relocation using a relocation register.

concept of a logical address space that is bound to a separate physical address space is central to proper memory management.

### 9.1.4 Dynamic Loading

In our discussion so far, it has been necessary for the entire program and all data of a process to be in physical memory for the process to execute. The size of a process has thus been limited to the size of physical memory. To obtain better memory-space utilization, we can use **dynamic loading**. With dynamic loading, a routine is not loaded until it is called. All routines are kept on disk in a relocatable load format. The main program is loaded into memory and is executed. When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded. If it has not, the relocatable linking loader is called to load the desired routine into memory and to update the program's address tables to reflect this change. Then control is passed to the newly loaded routine.

The advantage of dynamic loading is that a routine is loaded only when it is needed. This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines. In such a situation, although the total program size may be large, the portion that is used (and hence loaded) may be much smaller.

Dynamic loading does not require special support from the operating system. It is the responsibility of the users to design their programs to take advantage of such a method. Operating systems may help the programmer, however, by providing library routines to implement dynamic loading.

### 9.1.5 Dynamic Linking and Shared Libraries

**Dynamically linked libraries** (**DLLs**) are system libraries that are linked to user programs when the programs are run (refer back to Figure 9.3). Some operating systems support only **static linking**, in which system libraries are treated like any other object module and are combined by the loader into the binary program image. Dynamic linking, in contrast, is similar to dynamic loading. Here, though, linking, rather than loading, is postponed until execution time. This feature is usually used with system libraries, such as the standard C language library. Without this facility, each program on a system must include a copy of its language library (or at least the routines referenced by the program) in the executable image. This requirement not only increases the size of an executable image but also may waste main memory. A second advantage of DLLs is that these libraries can be shared among multiple processes, so that only one instance of the DLL in main memory. For this reason, DLLs are also known as **shared libraries**, and are used extensively in Windows and Linux systems.

When a program references a routine that is in a dynamic library, the loader locates the DLL, loading it into memory if necessary. It then adjusts addresses that reference functions in the dynamic library to the location in memory where the DLL is stored.

Dynamically linked libraries can be extended to library updates (such as bug fixes). In addition, a library may be replaced by a new version, and all programs that reference the library will automatically use the new version. Without dynamic linking, all such programs would need to be relinked to gain access to the new library. So that programs will not accidentally execute new, incompatible versions of libraries, version information is included in both the program and the library. More than one version of a library may be loaded into memory, and each program uses its version information to decide which copy of the library to use. Versions with minor changes retain the same version number, whereas versions with major changes increment the number. Thus, only programs that are compiled with the new library version are affected by any incompatible changes incorporated in it. Other programs linked before the new library was installed will continue using the older library.

Unlike dynamic loading, dynamic linking and shared libraries generally require help from the operating system. If the processes in memory are protected from one another, then the operating system is the only entity that can check to see whether the needed routine is in another process's memory space or that can allow multiple processes to access the same memory addresses. We elaborate on this concept, as well as how DLLs can be shared by multiple processes, when we discuss paging in Section 9.3.4.

## 9.2 Contiguous Memory Allocation

The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate main memory in the most efficient way possible. This section explains one early method, contiguous memory allocation.

The memory is usually divided into two partitions: one for the operating system and one for the user processes. We can place the operating system in either low memory addresses or high memory addresses. This decision depends on many factors, such as the location of the interrupt vector. However, many operating systems (including Linux and Windows) place the operating system in high memory, and therefore we discuss only that situation.

We usually want several user processes to reside in memory at the same time. We therefore need to consider how to allocate available memory to the processes that are waiting to be brought into memory. In **contiguous memory allocation**, each process is contained in a single section of memory that is contiguous to the section containing the next process. Before discussing this memory allocation scheme further, though, we must address the issue of memory protection.

### 9.2.1 Memory Protection

We can prevent a process from accessing memory that it does not own by combining two ideas previously discussed. If we have a system with a relocation register (Section 9.1.3), together with a limit register (Section 9.1.1), we accomplish our goal. The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses (for example, relocation = 100040 and limit = 74600). Each logical address must fall within the range specified by the limit register. The MMU maps the logical address dynamically by adding the value in the relocation register. This mapped address is sent to memory (Figure 9.6).

When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. Because every address generated by a CPU is checked against these registers, we can protect both the operating system and the other users' programs and data from being modified by this running process.

The relocation-register scheme provides an effective way to allow the operating system's size to change dynamically. This flexibility is desirable in many situations. For example, the operating system contains code and buffer space for device drivers. If a device driver is not currently in use, it makes little sense to keep it in memory; instead, it can be loaded into memory only when it is needed. Likewise, when the device driver is no longer needed, it can be removed and its memory allocated for other needs.

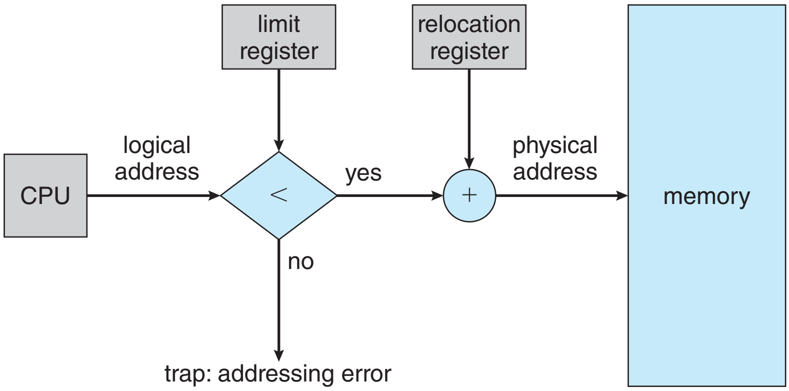


**Figure 9.6** Hardware support for relocation and limit registers.

### 9.2.2 Memory Allocation

Now we are ready to turn to memory allocation. One of the simplest methods of allocating memory is to assign processes to variably sized partitions in memory, where each partition may contain exactly one process. In this **variable-partition** scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes and is considered one large block of available memory, a **hole**. Eventually, as you will see, memory contains a set of holes of various sizes.

Figure 9.7 depicts this scheme. Initially, the memory is fully utilized, containing processes 5, 8, and 2. After process 8 leaves, there is one contiguous hole. Later on, process 9 arrives and is allocated memory. Then process 5 departs, resulting in two noncontiguous holes.

As processes enter the system, the operating system takes into account the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory. When a process is allocated space, it is loaded into memory, where it can then compete for CPU time. When a process terminates, it releases its memory, which the operating system may then provide to another process.

What happens when there isn't sufficient memory to satisfy the demands of an arriving process? One option is to simply reject the process and provide an appropriate error message. Alternatively, we can place such processes into a wait queue. When memory is later released, the operating system checks the wait queue to determine if it will satisfy the memory demands of a waiting process.

In general, as mentioned, the memory blocks available comprise a *set* of holes of various sizes scattered throughout memory. When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process. If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole.

This procedure is a particular instance of the general **dynamic storage-allocation problem**, which concerns how to satisfy a request of size *n* from a list of free holes. There are many solutions to this problem. The **first-fit**, **best-fit**, and **worst-fit** strategies are the ones most commonly used to select a free hole from the set of available holes.

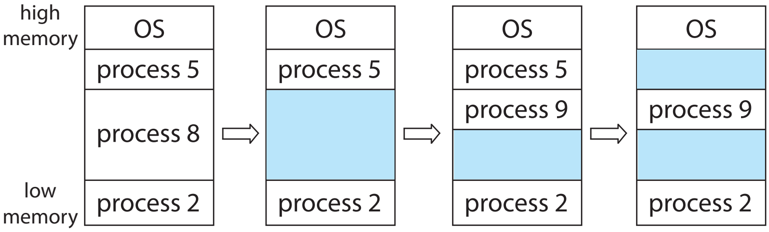


**Figure 9.7** Variable partition.

- **First fit**. Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
- **Best fit**. Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.
- **Worst fit**. Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Simulations have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization. Neither first fit nor best fit is clearly better than the other in terms of storage utilization, but first fit is generally faster.

### 9.2.3 Fragmentation

Both the first-fit and best-fit strategies for memory allocation suffer from **external fragmentation**. As processes are loaded and removed from memory, the free memory space is broken into little pieces. External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous: storage is fragmented into a large number of small holes. This fragmentation problem can be severe. In the worst case, we could have a block of free (or wasted) memory between every two processes. If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.

Whether we are using the first-fit or best-fit strategy can affect the amount of fragmentation. (First fit is better for some systems, whereas best fit is better for others.) Another factor is which end of a free block is allocated. (Which is the leftover piece—the one on the top or the one on the bottom?) No matter which algorithm is used, however, external fragmentation will be a problem.

Depending on the total amount of memory storage and the average process size, external fragmentation may be a minor or a major problem. Statistical analysis of first fit, for instance, reveals that, even with some optimization, given $N$ allocated blocks, another 0.5 $N$ blocks will be lost to fragmentation. That is, one-third of memory may be unusable! This property is known as the **50-percent rule**.

Memory fragmentation can be internal as well as external. Consider a multiple-partition allocation scheme with a hole of 18,464 bytes. Suppose that the next process requests 18,462 bytes. If we allocate exactly the requested block, we are left with a hole of 2 bytes. The overhead to keep track of this hole will be substantially larger than the hole itself. The general approach to avoiding this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size. With this approach, the memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is **internal fragmentation**—unused memory that is internal to a partition.

One solution to the problem of external fragmentation is **compaction**. The goal is to shuffle the memory contents so as to place all free memory together in one large block. Compaction is not always possible, however. If relocation is static and is done at assembly or load time, compaction cannot be done. It is possible only if relocation is dynamic and is done at execution time. If addresses are relocated dynamically, relocation requires only moving the program and data and then changing the base register to reflect the new base address. When compaction is possible, we must determine its cost. The simplest compaction algorithm is to move all processes toward one end of memory; all holes move in the other direction, producing one large hole of available memory. This scheme can be expensive.

Another possible solution to the external-fragmentation problem is to permit the logical address space of processes to be noncontiguous, thus allowing a process to be allocated physical memory wherever such memory is available. This is the strategy used in ***paging,*** the most common memory-management technique for computer systems. We describe paging in the following section.

Fragmentation is a general problem in computing that can occur wherever we must manage blocks of data. We discuss the topic further in the storage management chapters (Chapter 11 through Chapter 15).

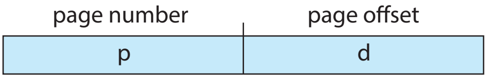
## 9.3 Paging

Memory management discussed thus far has required the physical address space of a process to be contiguous. We now introduce **paging**, a memory-management scheme that permits a process's physical address space to be noncontiguous. Paging avoids external fragmentation and the associated need for compaction, two problems that plague contiguous memory allocation. Because it offers numerous advantages, paging in its various forms is used in most operating systems, from those for large servers through those for mobile devices. Paging is implemented through cooperation between the operating system and the computer hardware.

### 9.3.1 Basic Method

The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called **frames** and breaking logical memory into blocks of the same size called **pages**. When a process is to be executed, its pages are loaded into any available memory frames from their source (a file system or the backing store). The backing store is divided into fixed-sized blocks that are the same size as the memory frames or clusters of multiple frames. This rather simple idea has great functionality and wide ramifications. For example, the logical address space is now totally separate from the physical address space, so a process can have a logical 64-bit address space even though the system has less than $2^{64}$ bytes of physical memory.

Every address generated by the CPU is divided into two parts: a **page number** (**p**) and a **page offset** (**d**):



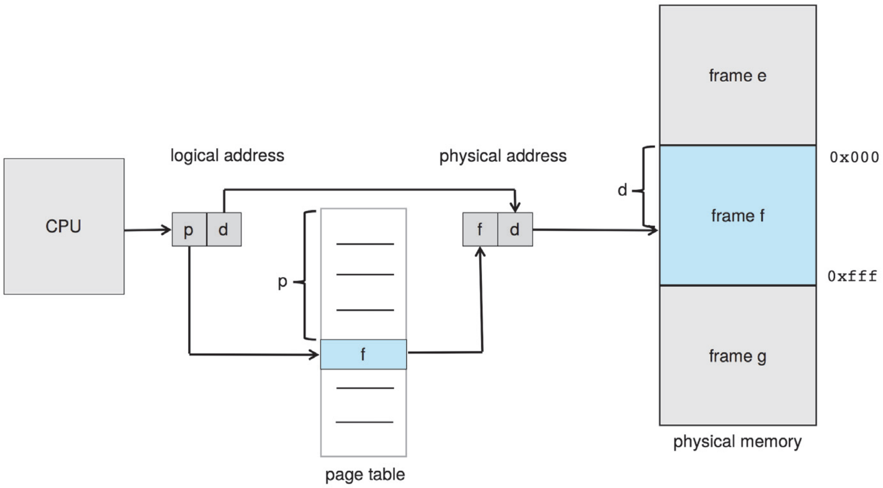


**Figure 9.8** Paging hardware.

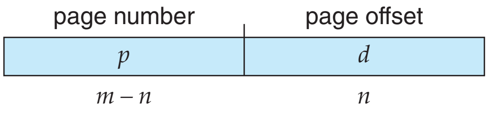
The page number is used as an index into a per-process **page table**. This is illustrated in Figure 9.8. The page table contains the base address of each frame in physical memory, and the offset is the location in the frame being referenced. Thus, the base address of the frame is combined with the page offset to define the physical memory address. The paging model of memory is shown in Figure 9.9.

The following outlines the steps taken by the MMU to translate a logical address generated by the CPU to a physical address:

1. Extract the page number $p$ and use it as an index into the page table.
2. Extract the corresponding frame number $f$ from the page table.
3. Replace the page number $p$ in the logical address with the frame number $f$.

As the offset $d$ does not change, it is not replaced, and the frame number and offset now comprise the physical address.

The page size (like the frame size) is defined by the hardware. The size of a page is a power of 2, typically varying between 4 KB and 1 GB per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy. If the size of the logical address space is $2^m$, and a page size is $2^n$ bytes, then the high-order $m-n$ bits of a logical address designate the page number, and the $n$ low-order bits designate the page offset. Thus, the logical address is as follows:



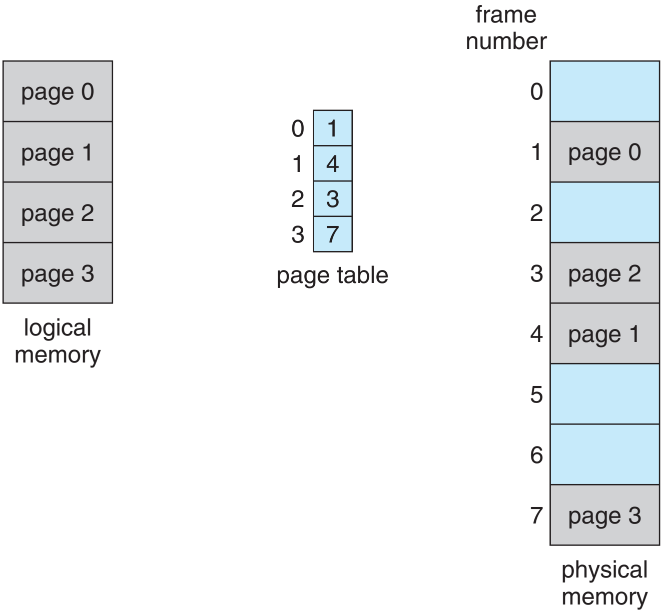


**Figure 9.9** Paging model of logical and physical memory.

where $p$ is an index into the page table and $d$ is the displacement within the page.

As a concrete (although minuscule) example, consider the memory in Figure 9.10. Here, in the logical address, $n = 2$ and $m = 4$. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the programmer's view of memory can be mapped into physical memory. Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 [= (5 × 4) + 0]. Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 × 4) + 3]. Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 [= (6 × 4) + 0]. Logical address 13 maps to physical address 9.

You may have noticed that paging itself is a form of dynamic relocation. Every logical address is bound by the paging hardware to some physical address. Using paging is similar to using a table of base (or relocation) registers, one for each frame of memory.

When we use a paging scheme, we have no external fragmentation: any free frame can be allocated to a process that needs it. However, we may have some internal fragmentation. Notice that frames are allocated as units. If the memory requirements of a process do not happen to coincide with page boundaries, the last frame allocated may not be completely full. For example, if page size is 2,048 bytes, a process of 72,766 bytes will need 35 pages plus 1,086 bytes. It will be allocated 36 frames, resulting in internal fragmentation of 2,048 − 1,086 = 962 bytes. In the worst case, a process would need $n$ pages plus 1 byte. It would be allocated $n + 1$ frames, resulting in internal fragmentation of almost an entire frame.

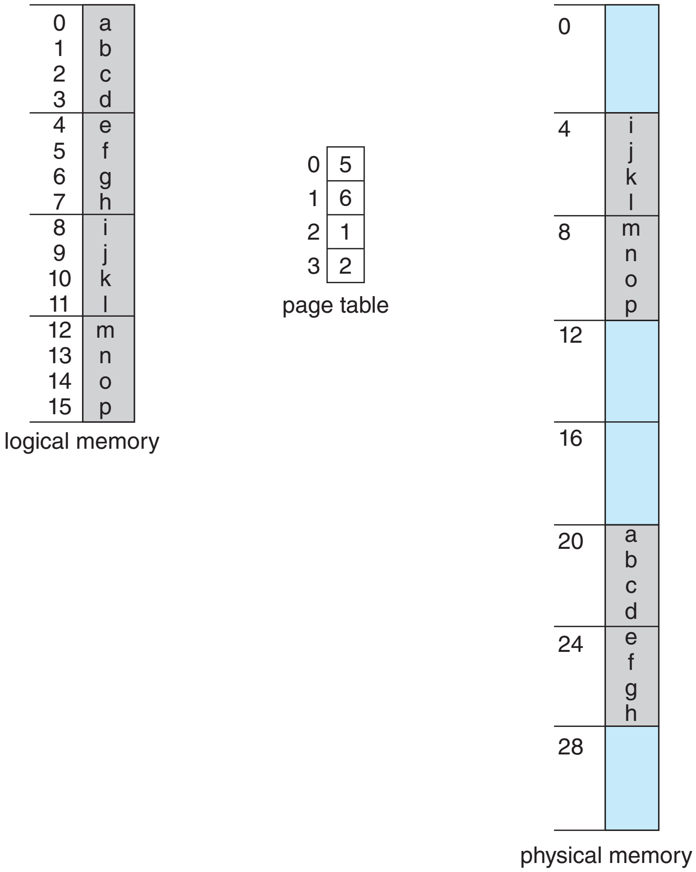


**Figure 9.10** Paging example for a 32-byte memory with 4-byte pages.

If process size is independent of page size, we expect internal fragmentation to average one-half page per process. This consideration suggests that small page sizes are desirable. However, overhead is involved in each page-table entry, and this overhead is reduced as the size of the pages increases. Also, disk I/O is more efficient when the amount of data being transferred is larger (Chapter 11). Generally, page sizes have grown over time as processes, data sets, and main memory have become larger. Today, pages are typically either 4 KB or 8 KB in size, and some systems support even larger page sizes. Some CPUs and operating systems even support multiple page sizes. For instance, on x86-64 systems, Windows 10 supports page sizes of 4 KB and 2 MB. Linux also supports two page sizes: a default page size (typically 4 KB) and an architecture-dependent larger page size called **huge pages**.

Frequently, on a 32-bit CPU, each page-table entry is 4 bytes long, but that size can vary as well. A 32-bit entry can point to one of $2^{32}$ physical page frames. If the frame size is 4 KB ($2^{12}$), then a system with 4-byte entries can address $2^{44}$ bytes (or 16 TB) of physical memory. We should note here that the size of physical memory in a paged memory system is typically different from the maximum logical size of a process. As we further explore paging, we will

### *OBTAINING THE PAGE SIZE ON LINUX SYSTEMS*

On a Linux system, the page size varies according to architecture, and there are several ways of obtaining the page size. One approach is to use the system call `getpagesize()`. Another strategy is to enter the following command on the command line:

```
getconf PAGESIZE
```

Each of these techniques returns the page size as a number of bytes.

introduce other information that must be kept in the page-table entries. That information reduces the number of bits available to address page frames. Thus, a system with 32-bit page-table entries may address less physical memory than the possible maximum.

When a process arrives in the system to be executed, its size, expressed in pages, is examined. Each page of the process needs one frame. Thus, if the process requires *n* pages, at least *n* frames must be available in memory. If *n* frames are available, they are allocated to this arriving process. The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process. The next page is loaded into another frame, its frame number is put into the page table, and so on (Figure 9.11).

An important aspect of paging is the clear separation between the programmer's view of memory and the actual physical memory. The programmer views memory as one single space, containing only this one program. In fact, the user program is scattered throughout physical memory, which also holds

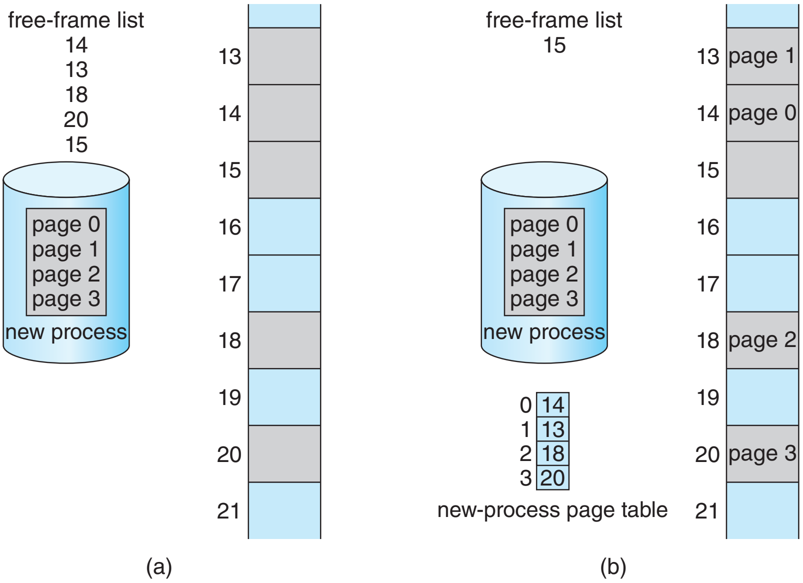

free-frame list
14
13
18
20
15
page 0
page 1
page 2
page 3
new process
13
14
15
16
17
18
19
20
21
(a)
free-frame list
15
page 0
page 1
page 2
page 3
new process
0 14
1 13
2 18
3 20
new-process page table
13 page 1
14 page 0
15
16
17
18 page 2
19
20 page 3
21
(b)

**Figure 9.11** Free frames (a) before allocation and (b) after allocation.

other programs. The difference between the programmer's view of memory and the actual physical memory is reconciled by the address-translation hardware. The logical addresses are translated into physical addresses. This mapping is hidden from the programmer and is controlled by the operating system. Notice that the user process by definition is unable to access memory it does not own. It has no way of addressing memory outside of its page table, and the table includes only those pages that the process owns.

Since the operating system is managing physical memory, it must be aware of the allocation details of physical memory—which frames are allocated, which frames are available, how many total frames there are, and so on. This information is generally kept in a single, system-wide data structure called a **frame table**. The frame table has one entry for each physical page frame, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process (or processes).

In addition, the operating system must be aware that user processes operate in user space, and all logical addresses must be mapped to produce physical addresses. If a user makes a system call (to do I/O, for example) and provides an address as a parameter (a buffer, for instance), that address must be mapped to produce the correct physical address. The operating system maintains a copy of the page table for each process, just as it maintains a copy of the instruction counter and register contents. This copy is used to translate logical addresses to physical addresses whenever the operating system must map a logical address to a physical address manually. It is also used by the CPU dispatcher to define the hardware page table when a process is to be allocated the CPU. Paging therefore increases the context-switch time.

### 9.3.2 Hardware Support

As page tables are per-process data structures, a pointer to the page table is stored with the other register values (like the instruction pointer) in the process control block of each process. When the CPU scheduler selects a process for execution, it must reload the user registers and the appropriate hardware page-table values from the stored user page table.

The hardware implementation of the page table can be done in several ways. In the simplest case, the page table is implemented as a set of dedicated high-speed hardware registers, which makes the page-address translation very efficient. However, this approach increases context-switch time, as each one of these registers must be exchanged during a context switch.

The use of registers for the page table is satisfactory if the page table is reasonably small (for example, 256 entries). Most contemporary CPUs, however, support much larger page tables (for example, $2^{20}$ entries). For these machines, the use of fast registers to implement the page table is not feasible. Rather, the page table is kept in main memory, and a **page-table base register** (**PTBR**) points to the page table. Changing page tables requires changing only this one register, substantially reducing context-switch time.

#### 9.3.2.1 Translation Look-Aside Buffer

Although storing the page table in main memory can yield faster context switches, it may also result in slower memory access times. Suppose we want to access location *i*. We must first index into the page table, using the value in the PTBR offset by the page number for *i*. This task requires one memory access. It provides us with the frame number, which is combined with the page offset to produce the actual address. We can then access the desired place in memory. With this scheme, ***two*** memory accesses are needed to access data (one for the page-table entry and one for the actual data). Thus, memory access is slowed by a factor of 2, a delay that is considered intolerable under most circumstances.

The standard solution to this problem is to use a special, small, fast-lookup hardware cache called a **translation look-aside buffer** (**TLB**). The TLB is associative, high-speed memory. Each entry in the TLB consists of two parts: a key (or tag) and a value. When the associative memory is presented with an item, the item is compared with all keys simultaneously. If the item is found, the corresponding value field is returned. The search is fast; a TLB lookup in modern hardware is part of the instruction pipeline, essentially adding no performance penalty. To be able to execute the search within a pipeline step, however, the TLB must be kept small. It is typically between 32 and 1,024 entries in size. Some CPUs implement separate instruction and data address TLBs. That can double the number of TLB entries available, because those lookups occur in different pipeline steps. We can see in this development an example of the evolution of CPU technology: systems have evolved from having no TLBs to having multiple levels of TLBs, just as they have multiple levels of caches.

The TLB is used with page tables in the following way. The TLB contains only a few of the page-table entries. When a logical address is generated by the CPU, the MMU first checks if its page number is present in the TLB. If the page number is found, its frame number is immediately available and is used to access memory. As just mentioned, these steps are executed as part of the instruction pipeline within the CPU, adding no performance penalty compared with a system that does not implement paging.

If the page number is not in the TLB (known as a **TLB miss**), address translation proceeds following the steps illustrated in Section 9.3.1, where a memory reference to the page table must be made. When the frame number is obtained, we can use it to access memory (Figure 9.12). In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference.

If the TLB is already full of entries, an existing entry must be selected for replacement. Replacement policies range from least recently used (LRU) through round-robin to random. Some CPUs allow the operating system to participate in LRU entry replacement, while others handle the matter themselves. Furthermore, some TLBs allow certain entries to be **wired down**, meaning that they cannot be removed from the TLB. Typically, TLB entries for key kernel code are wired down.

Some TLBs store **address-space identifiers** (**ASIDs**) in each TLB entry. An ASID uniquely identifies each process and is used to provide address-space protection for that process. When the TLB attempts to resolve virtual page numbers, it ensures that the ASID for the currently running process matches the ASID associated with the virtual page. If the ASIDs do not match, the attempt is treated as a TLB miss. In addition to providing address-space protection, an ASID allows the TLB to contain entries for several different processes simultaneously. If the TLB does not support separate ASIDs, then every time a new page table is selected (for instance, with each context switch), the TLB must be **flushed** (or erased) to ensure that the next executing process does not use the

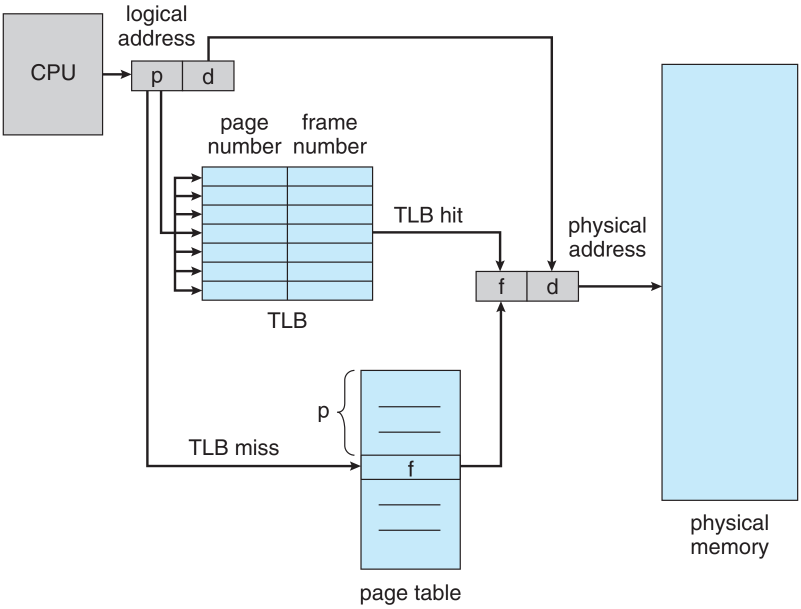


**Figure 9.12** Paging hardware with TLB.

wrong translation information. Otherwise, the TLB could include old entries that contain valid virtual addresses but have incorrect or invalid physical addresses left over from the previous process.

The percentage of times that the page number of interest is found in the TLB is called the **hit ratio**. An 80-percent hit ratio, for example, means that we find the desired page number in the TLB 80 percent of the time. If it takes 10 nanoseconds to access memory, then a mapped-memory access takes 10 nanoseconds when the page number is in the TLB. If we fail to find the page number in the TLB then we must first access memory for the page table and frame number (10 nanoseconds) and then access the desired byte in memory (10 nanoseconds), for a total of 20 nanoseconds. (We are assuming that a page-table lookup takes only one memory access, but it can take more, as we shall see.) To find the **effective memory-access time**, we weight the case by its probability:

$$\begin{aligned}\text{effective access time} &= 0.80 \times 10 + 0.20 \times 20\\ &= 12 \text{ nanoseconds}\end{aligned}$$

In this example, we suffer a 20-percent slowdown in average memory-access time (from 10 to 12 nanoseconds). For a 99-percent hit ratio, which is much more realistic, we have

$$\begin{aligned}\text{effective access time} &= 0.99 \times 10 + 0.01 \times 20\\ &= 10.1 \text{ nanoseconds}\end{aligned}$$

This increased hit rate produces only a 1 percent slowdown in access time.

As noted earlier, CPUs today may provide multiple levels of TLBs. Calculating memory access times in modern CPUs is therefore much more complicated than shown in the example above. For instance, the Intel Core i7 CPU has a 128-entry L1 instruction TLB and a 64-entry L1 data TLB. In the case of a miss at L1, it takes the CPU six cycles to check for the entry in the L2 512-entry TLB. A miss in L2 means that the CPU must either walk through the page-table entries in memory to find the associated frame address, which can take hundreds of cycles, or interrupt to the operating system to have it do the work.

A complete performance analysis of paging overhead in such a system would require miss-rate information about each TLB tier. We can see from the general information above, however, that hardware features can have a significant effect on memory performance and that operating-system improvements (such as paging) can result in and, in turn, be affected by hardware changes (such as TLBs). We will further explore the impact of the hit ratio on the TLB in Chapter 10.

TLBs are a hardware feature and therefore would seem to be of little concern to operating systems and their designers. But the designer needs to understand the function and features of TLBs, which vary by hardware platform. For optimal operation, an operating-system design for a given platform must implement paging according to the platform's TLB design. Likewise, a change in the TLB design (for example, between different generations of Intel CPUs) may necessitate a change in the paging implementation of the operating systems that use it.

### 9.3.3 Protection

Memory protection in a paged environment is accomplished by protection bits associated with each frame. Normally, these bits are kept in the page table.

One bit can define a page to be read–write or read-only. Every reference to memory goes through the page table to find the correct frame number. At the same time that the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read-only page. An attempt to write to a read-only page causes a hardware trap to the operating system (or memory-protection violation).

We can easily expand this approach to provide a finer level of protection. We can create hardware to provide read-only, read–write, or execute-only protection; or, by providing separate protection bits for each kind of access, we can allow any combination of these accesses. Illegal attempts will be trapped to the operating system.

One additional bit is generally attached to each entry in the page table: a **valid–invalid** bit. When this bit is set to *valid*, the associated page is in the process's logical address space and is thus a legal (or valid) page. When the bit is set to *invalid*, the page is not in the process's logical address space. Illegal addresses are trapped by use of the valid–invalid bit. The operating system sets this bit for each page to allow or disallow access to the page.

Suppose, for example, that in a system with a 14-bit address space (0 to 16383), we have a program that should use only addresses 0 to 10468. Given a page size of 2 KB, we have the situation shown in Figure 9.13. Addresses in pages 0, 1, 2, 3, 4, and 5 are mapped normally through the page table. Any attempt to generate an address in pages 6 or 7, however, will find that the

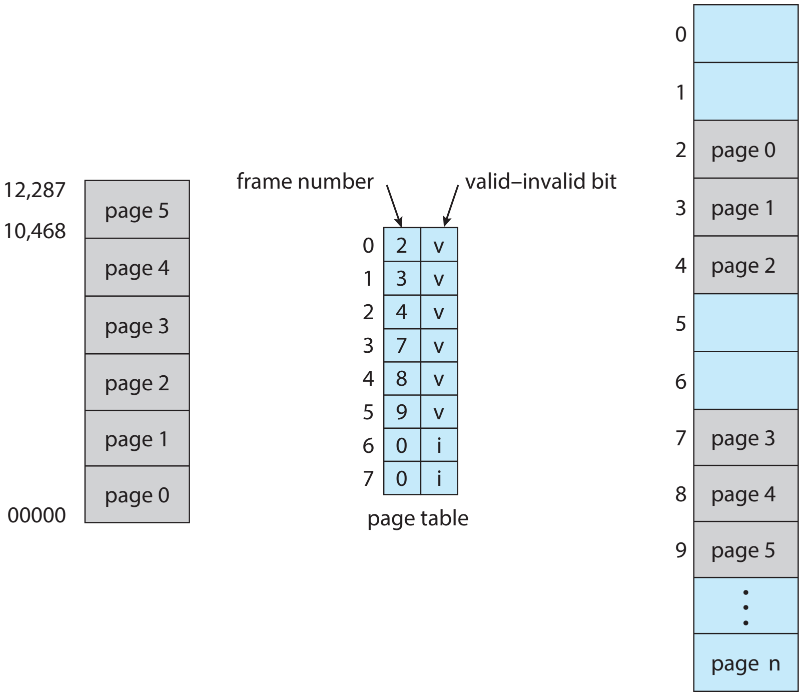

12,287
10,468
00000
page 5
page 4
page 3
page 2
page 1
page 0
frame number
valid–invalid bit
0 2 v
1 3 v
2 4 v
3 7 v
4 8 v
5 9 v
6 0 i
7 0 i
page table
0
1
2 page 0
3 page 1
4 page 2
5
6
7 page 3
8 page 4
9 page 5
⋮
page n

**Figure 9.13** Valid (v) or invalid (i) bit in a page table.

valid–invalid bit is set to invalid, and the computer will trap to the operating system (invalid page reference).

Notice that this scheme has created a problem. Because the program extends only to address 10468, any reference beyond that address is illegal. However, references to page 5 are classified as valid, so accesses to addresses up to 12287 are valid. Only the addresses from 12288 to 16383 are invalid. This problem is a result of the 2-KB page size and reflects the internal fragmentation of paging.

Rarely does a process use all its address range. In fact, many processes use only a small fraction of the address space available to them. It would be wasteful in these cases to create a page table with entries for every page in the address range. Most of this table would be unused but would take up valuable memory space. Some systems provide hardware, in the form of a **page-table length register** (**PTLR**), to indicate the size of the page table. This value is checked against every logical address to verify that the address is in the valid range for the process. Failure of this test causes an error trap to the operating system.

### 9.3.4 Shared Pages

An advantage of paging is the possibility of ***sharing*** common code, a consideration that is particularly important in an environment with multiple processes. Consider the standard C library, which provides a portion of the system call interface for many versions of UNIX and Linux. On a typical Linux system, most user processes require the standard C library `libc`. One option is to have each process load its own copy of `libc` into its address space. If a system has 40 user processes, and the `libc` library is 2 MB, this would require 80 MB of memory.

If the code is **reentrant code**, however, it can be shared, as shown in Figure 9.14. Here, we see three processes sharing the pages for the standard C library `libc`. (Although the figure shows the `libc` library occupying four pages, in reality, it would occupy more.) Reentrant code is non-self-modifying code: it never changes during execution. Thus, two or more processes can execute the same code at the same time. Each process has its own copy of registers and data storage to hold the data for the process's execution. The data for two different processes will, of course, be different. Only one copy of the standard C library need be kept in physical memory, and the page table for each user process maps onto the same physical copy of `libc`. Thus, to support 40 processes, we need only one copy of the library, and the total space now required is 2 MB instead of 80 MB—a significant saving!

In addition to run-time libraries such as `libc`, other heavily used programs can also be shared—compilers, window systems, database systems, and so on. The shared libraries discussed in Section 9.1.5 are typically implemented with shared pages. To be sharable, the code must be reentrant. The read-only nature of shared code should not be left to the correctness of the code; the operating system should enforce this property.

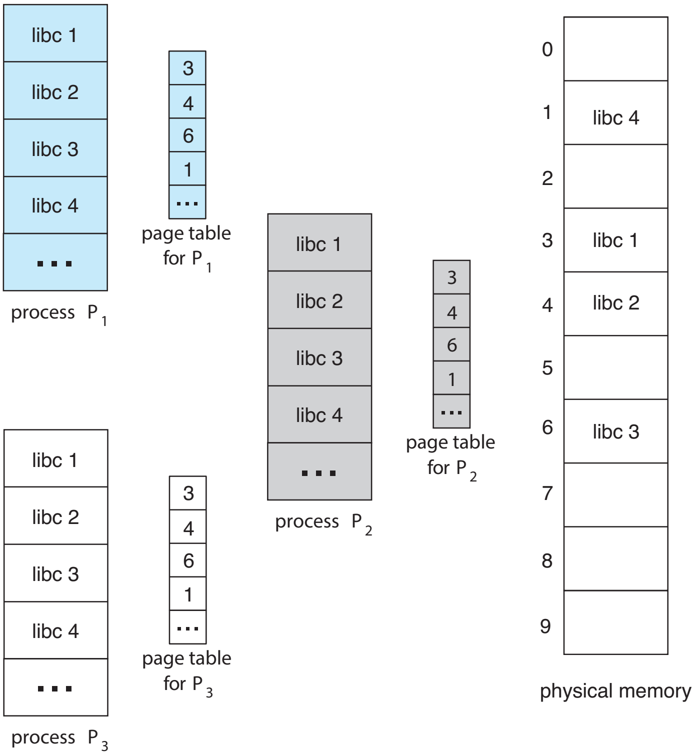

libc 1
libc 2
libc 3
libc 4
...
process P1
3
4
6
1
...
page table for P1
libc 1
libc 2
libc 3
libc 4
...
process P2
3
4
6
1
...
page table for P2
libc 1
libc 2
libc 3
libc 4
...
process P3
3
4
6
1
...
page table for P3
0
1 libc 4
2
3 libc 1
4 libc 2
5
6 libc 3
7
8
9
physical memory

**Figure 9.14** Sharing of standard C library in a paging environment.

The sharing of memory among processes on a system is similar to the sharing of the address space of a task by threads, described in Chapter 4. Furthermore, recall that in Chapter 3 we described shared memory as a method of interprocess communication. Some operating systems implement shared memory using shared pages.

Organizing memory according to pages provides numerous benefits in addition to allowing several processes to share the same physical pages. We cover several other benefits in Chapter 10.

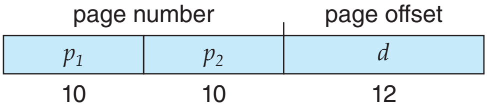
## 9.4 Structure of the Page Table

In this section, we explore some of the most common techniques for structuring the page table, including hierarchical paging, hashed page tables, and inverted page tables.

### 9.4.1 Hierarchical Paging

Most modern computer systems support a large logical address space ($2^{32}$ to $2^{64}$). In such an environment, the page table itself becomes excessively large. For example, consider a system with a 32-bit logical address space. If the page size in such a system is 4 KB ($2^{12}$), then a page table may consist of over 1 million entries ($2^{20} = 2^{32}/2^{12}$). Assuming that each entry consists of 4 bytes, each process may need up to 4 MB of physical address space for the page table alone. Clearly, we would not want to allocate the page table contiguously in main memory. One simple solution to this problem is to divide the page table into smaller pieces. We can accomplish this division in several ways.

One way is to use a two-level paging algorithm, in which the page table itself is also paged (Figure 9.15). For example, consider again the system with a 32-bit logical address space and a page size of 4 KB. A logical address is divided into a page number consisting of 20 bits and a page offset consisting of 12 bits. Because we page the page table, the page number is further divided into a 10-bit page number and a 10-bit page offset. Thus, a logical address is as follows:



where $p_1$ is an index into the outer page table and $p_2$ is the displacement within the page of the inner page table. The address-translation method for this architecture is shown in Figure 9.16. Because address translation works from the outer page table inward, this scheme is also known as a **forward-mapped** page table.

For a system with a 64-bit logical address space, a two-level paging scheme is no longer appropriate. To illustrate this point, let's suppose that the page size in such a system is 4 KB ($2^{12}$). In this case, the page table consists of up to $2^{52}$ entries. If we use a two-level paging scheme, then the inner page tables can conveniently be one page long, or contain $2^{10}$ 4-byte entries. The addresses look like this:

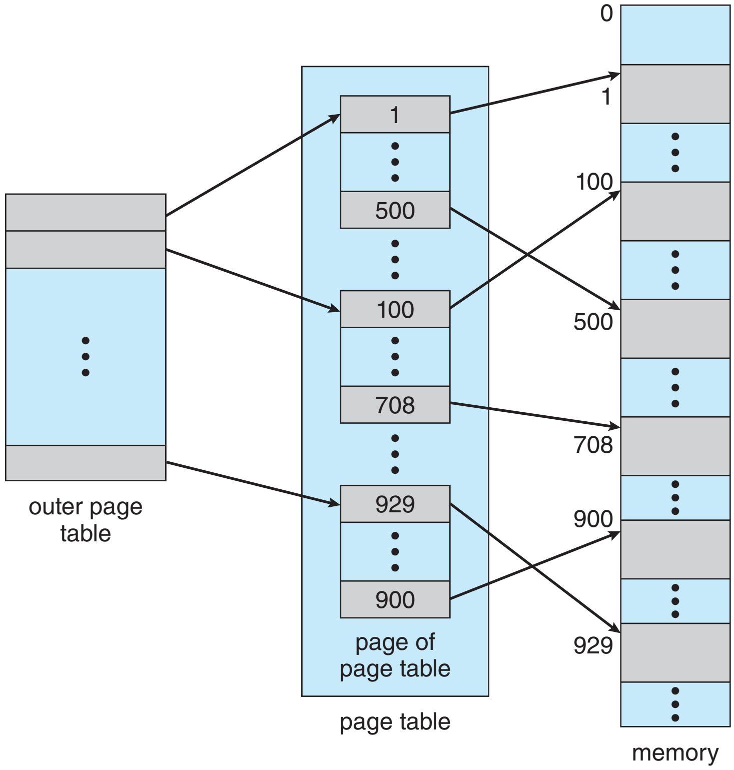


**Figure 9.15** A two-level page-table scheme.

| outer page | inner page | offset |
|---|---|---|
| $p_1$ | $p_2$ | $d$ |
| 42 | 10 | 12 |

The outer page table consists of $2^{42}$ entries, or $2^{44}$ bytes. The obvious way to avoid such a large table is to divide the outer page table into smaller pieces. (This approach is also used on some 32-bit processors for added flexibility and efficiency.)

We can divide the outer page table in various ways. For example, we can page the outer page table, giving us a three-level paging scheme. Suppose that the outer page table is made up of standard-size pages ($2^{10}$ entries, or $2^{12}$ bytes). In this case, a 64-bit address space is still daunting:

| 2nd outer page | outer page | inner page | offset |
|---|---|---|---|
| $p_1$ | $p_2$ | $p_3$ | $d$ |
| 32 | 10 | 10 | 12 |

The outer page table is still $2^{34}$ bytes (16 GB) in size.

The next step would be a four-level paging scheme, where the second-level outer page table itself is also paged, and so forth. The 64-bit UltraSPARC would require seven levels of paging—a prohibitive number of memory accesses—

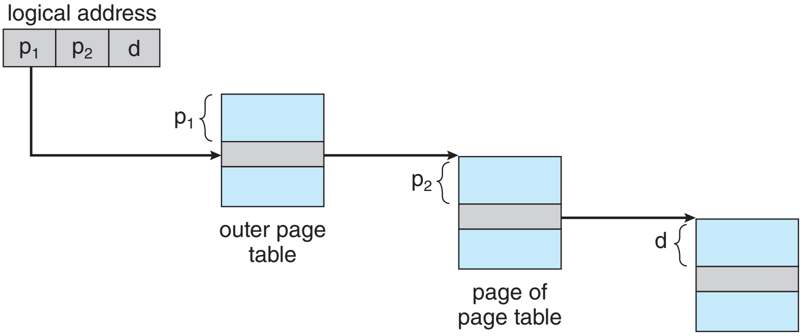

logical address
$p_1$
$p_2$
d
$p_1$
outer page table
$p_2$
page of page table
d

**Figure 9.16** Address translation for a two-level 32-bit paging architecture.

to translate each logical address. You can see from this example why, for 64-bit architectures, hierarchical page tables are generally considered inappropriate.

### 9.4.2 Hashed Page Tables

One approach for handling address spaces larger than 32 bits is to use a **hashed page table**, with the hash value being the virtual page number. Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions). Each element consists of three fields: (1) the virtual page number, (2) the value of the mapped page frame, and (3) a pointer to the next element in the linked list.

The algorithm works as follows: The virtual page number in the virtual address is hashed into the hash table. The virtual page number is compared with field 1 in the first element in the linked list. If there is a match, the corresponding page frame (field 2) is used to form the desired physical address. If there is no match, subsequent entries in the linked list are searched for a matching virtual page number. This scheme is shown in Figure 9.17.

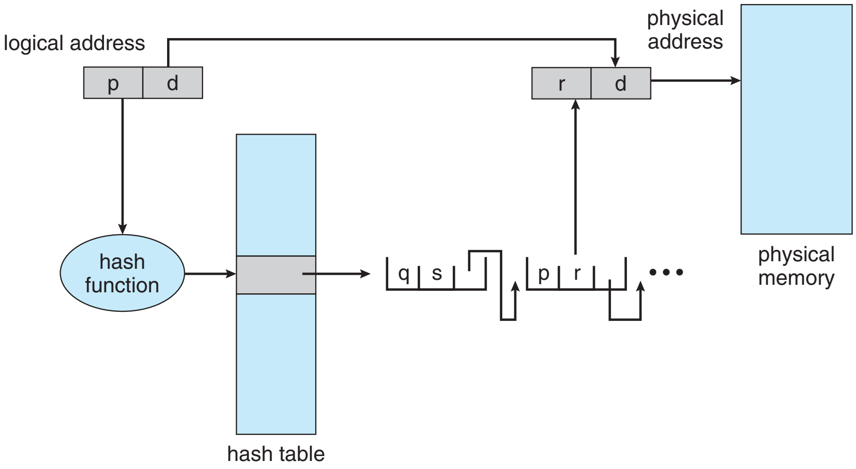

logical address
p
d
physical address
r
d
hash function
q
s
p
r
• • •
physical memory
hash table

**Figure 9.17** Hashed page table.

A variation of this scheme that is useful for 64-bit address spaces has been proposed. This variation uses **clustered page tables**, which are similar to hashed page tables except that each entry in the hash table refers to several pages (such as 16) rather than a single page. Therefore, a single page-table entry can store the mappings for multiple physical-page frames. Clustered page tables are particularly useful for **sparse** address spaces, where memory references are noncontiguous and scattered throughout the address space.

### 9.4.3 Inverted Page Tables

Usually, each process has an associated page table. The page table has one entry for each page that the process is using (or one slot for each virtual address, regardless of the latter's validity). This table representation is a natural one, since processes reference pages through the pages' virtual addresses. The operating system must then translate this reference into a physical memory address. Since the table is sorted by virtual address, the operating system is able to calculate where in the table the associated physical address entry is located and to use that value directly. One of the drawbacks of this method is that each page table may consist of millions of entries. These tables may consume large amounts of physical memory just to keep track of how other physical memory is being used.

To solve this problem, we can use an **inverted page table**. An inverted page table has one entry for each real page (or frame) of memory. Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns the page. Thus, only one page table is in the system, and it has only one entry for each page of physical memory. Figure 9.18 shows the operation of an inverted page table. Compare it with Figure 9.8, which depicts a standard page table in operation. Inverted page tables often require that an address-space identifier (Section 9.3.2) be stored in each entry of the page table, since the table usually contains several

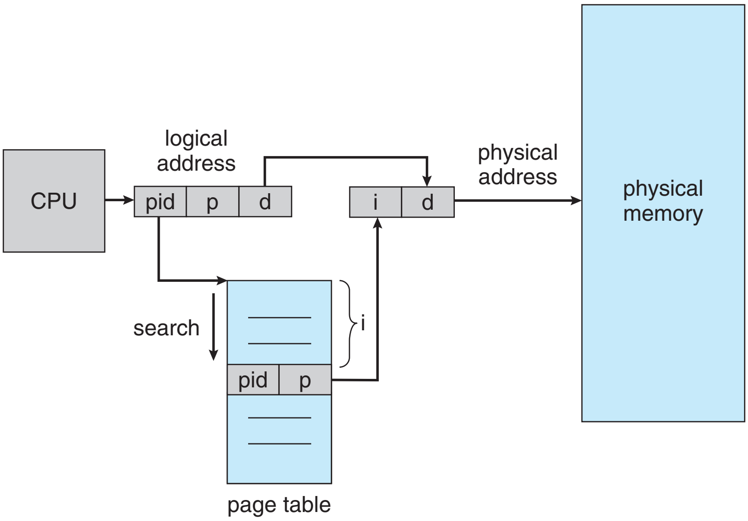


**Figure 9.18** Inverted page table.

different address spaces mapping physical memory. Storing the address-space identifier ensures that a logical page for a particular process is mapped to the corresponding physical page frame. Examples of systems using inverted page tables include the 64-bit UltraSPARC and PowerPC.

To illustrate this method, we describe a simplified version of the inverted page table used in the IBM RT. IBM was the first major company to use inverted page tables, starting with the IBM System 38 and continuing through the RS/6000 and the current IBM Power CPUs. For the IBM RT, each virtual address in the system consists of a triple:

<process-id, page-number, offset>.

Each inverted page-table entry is a pair <process-id, page-number> where the process-id assumes the role of the address-space identifier. When a memory reference occurs, part of the virtual address, consisting of <process-id, page-number>, is presented to the memory subsystem. The inverted page table is then searched for a match. If a match is found—say, at entry *i*—then the physical address <*i*, offset> is generated. If no match is found, then an illegal address access has been attempted.

Although this scheme decreases the amount of memory needed to store each page table, it increases the amount of time needed to search the table when a page reference occurs. Because the inverted page table is sorted by physical address, but lookups occur on virtual addresses, the whole table might need to be searched before a match is found. This search would take far too long. To alleviate this problem, we use a hash table, as described in Section 9.4.2, to limit the search to one—or at most a few—page-table entries. Of course, each access to the hash table adds a memory reference to the procedure, so one virtual memory reference requires at least two real memory reads—one for the hash-table entry and one for the page table. (Recall that the TLB is searched first, before the hash table is consulted, offering some performance improvement.)

One interesting issue with inverted page tables involves shared memory. With standard paging, each process has its own page table, which allows multiple virtual addresses to be mapped to the same physical address. This method cannot be used with inverted page tables; because there is only one virtual page entry for every physical page, one physical page cannot have two (or more) shared virtual addresses. Therefore, with inverted page tables, only one mapping of a virtual address to the shared physical address may occur at any given time. A reference by another process sharing the memory will result in a page fault and will replace the mapping with a different virtual address.

### 9.4.4 Oracle SPARC Solaris

Consider as a final example a modern 64-bit CPU and operating system that are tightly integrated to provide low-overhead virtual memory. **Solaris** running on the **SPARC** CPU is a fully 64-bit operating system and as such has to solve the problem of virtual memory without using up all of its physical memory by keeping multiple levels of page tables. Its approach is a bit complex but solves the problem efficiently using hashed page tables. There are two hash tables—one for the kernel and one for all user processes. Each maps memory addresses from virtual to physical memory. Each hash-table entry represents a contiguous area of mapped virtual memory, which is more efficient than having a separate hash-table entry for each page. Each entry has a base address and a span indicating the number of pages the entry represents.

Virtual-to-physical translation would take too long if each address required searching through a hash table, so the CPU implements a TLB that holds translation table entries (TTEs) for fast hardware lookups. A cache of these TTEs resides in a translation storage buffer (TSB), which includes an entry per recently accessed page. When a virtual address reference occurs, the hardware searches the TLB for a translation. If none is found, the hardware walks through the in-memory TSB looking for the TTE that corresponds to the virtual address that caused the lookup. This **TLB walk** functionality is found on many modern CPUs. If a match is found in the TSB, the CPU copies the TSB entry into the TLB, and the memory translation completes. If no match is found in the TSB, the kernel is interrupted to search the hash table. The kernel then creates a TTE from the appropriate hash table and stores it in the TSB for automatic loading into the TLB by the CPU memory-management unit. Finally, the interrupt handler returns control to the MMU, which completes the address translation and retrieves the requested byte or word from main memory.

## 9.5 Swapping

Process instructions and the data they operate on must be in memory to be executed. However, a process, or a portion of a process, can be **swapped** temporarily out of memory to a **backing store** and then brought back into memory for continued execution (Figure 9.19). Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system, thus increasing the degree of multiprogramming in a system.

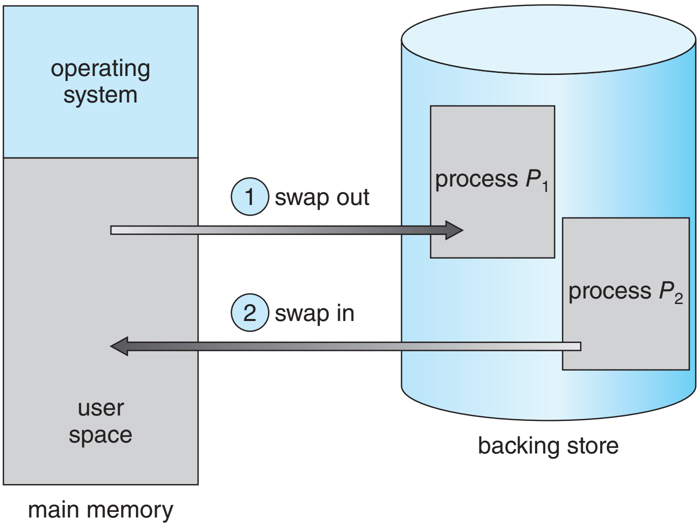


**Figure 9.19** Standard swapping of two processes using a disk as a backing store.

### 9.5.1 Standard Swapping

Standard swapping involves moving entire processes between main memory and a backing store. The backing store is commonly fast secondary storage. It must be large enough to accommodate whatever parts of processes need to be stored and retrieved, and it must provide direct access to these memory images. When a process or part is swapped to the backing store, the data structures associated with the process must be written to the backing store. For a multithreaded process, all per-thread data structures must be swapped as well. The operating system must also maintain metadata for processes that have been swapped out, so they can be restored when they are swapped back in to memory.

The advantage of standard swapping is that it allows physical memory to be oversubscribed, so that the system can accommodate more processes than there is actual physical memory to store them. Idle or mostly idle processes are good candidates for swapping; any memory that has been allocated to these inactive processes can then be dedicated to active processes. If an inactive process that has been swapped out becomes active once again, it must then be swapped back in. This is illustrated in Figure 9.19.

### 9.5.2 Swapping with Paging

Standard swapping was used in traditional UNIX systems, but it is generally no longer used in contemporary operating systems, because the amount of time required to move entire processes between memory and the backing store is prohibitive. (An exception to this is Solaris, which still uses standard swapping, however only under dire circumstances when available memory is extremely low.)

Most systems, including Linux and Windows, now use a variation of swapping in which pages of a process—rather than an entire process—can be swapped. This strategy still allows physical memory to be oversubscribed, but does not incur the cost of swapping entire processes, as presumably only a small number of pages will be involved in swapping. In fact, the term ***swapping*** now generally refers to standard swapping, and ***paging*** refers to swapping with paging. A **page out** operation moves a page from memory to the backing store; the reverse process is known as a **page in**. Swapping with paging is illustrated in Figure 9.20 where a subset of pages for processes *A* and *B* are being paged-out and paged-in respectively. As we shall see in Chapter 10, swapping with paging works well in conjunction with virtual memory.

### 9.5.3 Swapping on Mobile Systems

Most operating systems for PCs and servers support swapping pages. In contrast, mobile systems typically do not support swapping in any form. Mobile devices generally use flash memory rather than more spacious hard disks for nonvolatile storage. The resulting space constraint is one reason why mobile operating-system designers avoid swapping. Other reasons include the limited number of writes that flash memory can tolerate before it becomes unreliable and the poor throughput between main memory and flash memory in these devices.

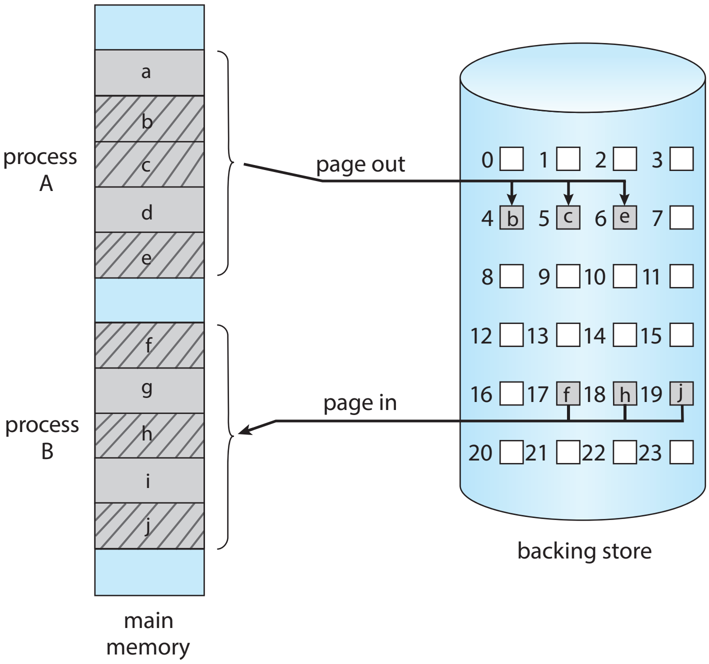


**Figure 9.20** Swapping with paging.

Instead of using swapping, when free memory falls below a certain threshold, Apple's iOS ***asks*** applications to voluntarily relinquish allocated memory. Read-only data (such as code) are removed from main memory and later reloaded from flash memory if necessary. Data that have been modified (such as the stack) are never removed. However, any applications that fail to free up sufficient memory may be terminated by the operating system.

Android adopts a strategy similar to that used by iOS. It may terminate a process if insufficient free memory is available. However, before terminating a process, Android writes its **application state** to flash memory so that it can be quickly restarted.

Because of these restrictions, developers for mobile systems must carefully allocate and release memory to ensure that their applications do not use too much memory or suffer from memory leaks.

### *SYSTEM PERFORMANCE UNDER SWAPPING*

Although swapping pages is more efficient than swapping entire processes, when a system is undergoing *any* form of swapping, it is often a sign there are more active processes than available physical memory. There are generally two approaches for handling this situation: (1) terminate some processes, or (2) get more physical memory!

## 9.6 Example: Intel 32- and 64-bit Architectures

The architecture of Intel chips has dominated the personal computer landscape for decades. The 16-bit Intel 8086 appeared in the late 1970s and was soon followed by another 16-bit chip—the Intel 8088—which was notable for being the chip used in the original IBM PC. Intel later produced a series of 32-bit chips—the IA-32—which included the family of 32-bit Pentium processors. More recently, Intel has produced a series of 64-bit chips based on the x86-64 architecture. Currently, all the most popular PC operating systems run on Intel chips, including Windows, macOS, and Linux (although Linux, of course, runs on several other architectures as well). Notably, however, Intel's dominance has not spread to mobile systems, where the ARM architecture currently enjoys considerable success (see Section 9.7).

In this section, we examine address translation for both IA-32 and x86-64 architectures. Before we proceed, however, it is important to note that because Intel has released several versions—as well as variations—of its architectures over the years, we cannot provide a complete description of the memory-management structure of all its chips. Nor can we provide all of the CPU details, as that information is best left to books on computer architecture. Rather, we present the major memory-management concepts of these Intel CPUs.

### 9.6.1 IA-32 Architecture

Memory management in IA-32 systems is divided into two components—segmentation and paging—and works as follows: The CPU generates logical addresses, which are given to the segmentation unit. The segmentation unit produces a linear address for each logical address. The linear address is then given to the paging unit, which in turn generates the physical address in main memory. Thus, the segmentation and paging units form the equivalent of the memory-management unit (MMU). This scheme is shown in Figure 9.21.

#### 9.6.1.1 IA-32 Segmentation

The IA-32 architecture allows a segment to be as large as 4 GB, and the maximum number of segments per process is 16 K. The logical address space of a process is divided into two partitions. The first partition consists of up to 8 K segments that are private to that process. The second partition consists of up to 8 K segments that are shared among all the processes. Information about the first partition is kept in the **local descriptor table** (LDT); information about the second partition is kept in the **global descriptor table** (GDT). Each entry in the LDT and GDT consists of an 8-byte segment descriptor with detailed information about a particular segment, including the base location and limit of that segment.

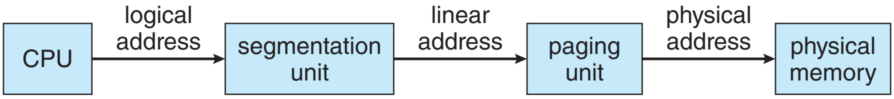


**Figure 9.21** Logical to physical address translation in IA-32.

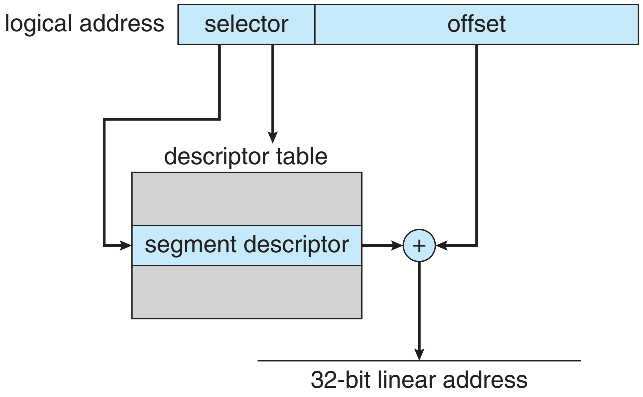


**Figure 9.22** IA-32 segmentation.

The logical address is a pair (selector, offset), where the selector is a 16-bit number:

| $s$ | $g$ | $p$ |
|---|---|---|
| 13 | 1 | 2 |

Here, $s$ designates the segment number, $g$ indicates whether the segment is in the GDT or LDT, and $p$ deals with protection. The offset is a 32-bit number specifying the location of the byte within the segment in question.

The machine has six segment registers, allowing six segments to be addressed at any one time by a process. It also has six 8-byte microprogram registers to hold the corresponding descriptors from either the LDT or the GDT. This cache lets the Pentium avoid having to read the descriptor from memory for every memory reference.

The linear address on the IA-32 is 32 bits long and is formed as follows. The segment register points to the appropriate entry in the LDT or GDT. The base and limit information about the segment in question is used to generate a **linear address**. First, the limit is used to check for address validity. If the address is not valid, a memory fault is generated, resulting in a trap to the operating system. If it is valid, then the value of the offset is added to the value of the base, resulting in a 32-bit linear address. This is shown in Figure 9.22. In the following section, we discuss how the paging unit turns this linear address into a physical address.

#### 9.6.1.2 IA-32 Paging

The IA-32 architecture allows a page size of either 4 KB or 4 MB. For 4-KB pages, IA-32 uses a two-level paging scheme in which the division of the 32-bit linear address is as follows:

| page number | | page offset |
|---|---|---|
| $p_1$ | $p_2$ | $d$ |
| 10 | 10 | 12 |

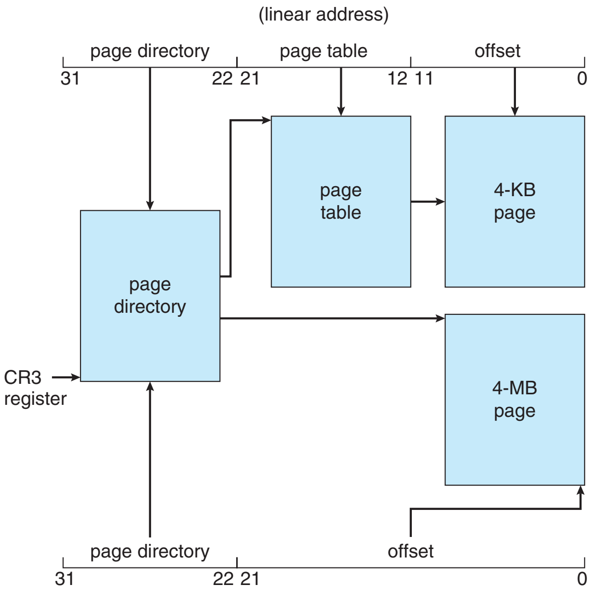


**Figure 9.23** Paging in the IA-32 architecture.

The address-translation scheme for this architecture is similar to the scheme shown in Figure 9.16. The IA-32 address translation is shown in more detail in Figure 9.23. The 10 high-order bits reference an entry in the outermost page table, which IA-32 terms the **page directory**. (The `CR3` register points to the page directory for the current process.) The page directory entry points to an inner page table that is indexed by the contents of the innermost 10 bits in the linear address. Finally, the low-order bits 0–11 refer to the offset in the 4-KB page pointed to in the page table.

One entry in the page directory is the `Page_Size` flag, which—if set—indicates that the size of the page frame is 4 MB and not the standard 4 KB. If this flag is set, the page directory points directly to the 4-MB page frame, bypassing the inner page table; and the 22 low-order bits in the linear address refer to the offset in the 4-MB page frame.

To improve the efficiency of physical memory use, IA-32 page tables can be swapped to disk. In this case, an invalid bit is used in the page directory entry to indicate whether the table to which the entry is pointing is in memory or on disk. If the table is on disk, the operating system can use the other 31 bits to specify the disk location of the table. The table can then be brought into memory on demand.

As software developers began to discover the 4-GB memory limitations of 32-bit architectures, Intel adopted a **page address extension** (**PAE**), which allows 32-bit processors to access a physical address space larger than 4 GB. The fundamental difference introduced by PAE support was that paging went from a two-level scheme (as shown in Figure 9.23) to a three-level scheme, where the top two bits refer to a **page directory pointer table**. Figure 9.24 illustrates a PAE system with 4-KB pages. (PAE also supports 2-MB pages.)

PAE also increased the page-directory and page-table entries from 32 to 64 bits in size, which allowed the base address of page tables and page frames to

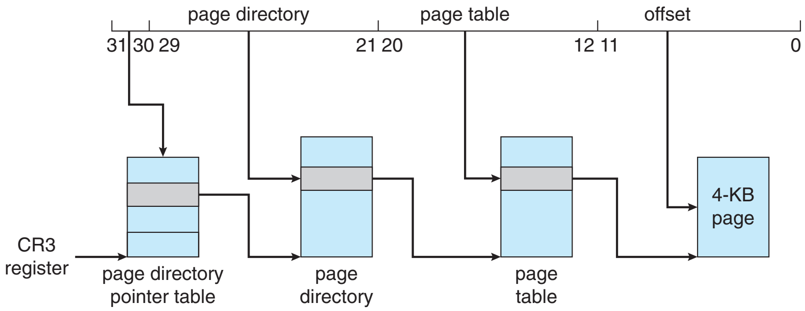

page directory
page table
offset
31
30 29
21 20
12 11
0
CR3
register
page directory
pointer table
page
directory
page
table
4-KB
page

**Figure 9.24** Page address extensions.

extend from 20 to 24 bits. Combined with the 12-bit offset, adding PAE support to IA-32 increased the address space to 36 bits, which supports up to 64 GB of physical memory. It is important to note that operating system support is required to use PAE. Both Linux and macOS support PAE. However, 32-bit versions of Windows desktop operating systems still provide support for only 4 GB of physical memory, even if PAE is enabled.

### 9.6.2 x86-64

Intel has had an interesting history of developing 64-bit architectures. Its initial entry was the IA-64 (later named **Itanium**) architecture, but that architecture was not widely adopted. Meanwhile, another chip manufacturer—AMD — began developing a 64-bit architecture known as x86-64 that was based on extending the existing IA-32 instruction set. The x86-64 supported much larger logical and physical address spaces, as well as several other architectural advances. Historically, AMD had often developed chips based on Intel's architecture, but now the roles were reversed as Intel adopted AMD's x86-64 architecture. In discussing this architecture, rather than using the commercial names **AMD64** and **Intel 64**, we will use the more general term **x86-64**.

Support for a 64-bit address space yields an astonishing $2^{64}$ bytes of addressable memory—a number greater than 16 quintillion (or 16 exabytes). However, even though 64-bit systems can potentially address this much memory, in practice far fewer than 64 bits are used for address representation in current designs. The x86-64 architecture currently provides a 48-bit virtual address with support for page sizes of 4 KB, 2 MB, or 1 GB using four levels of paging hierarchy. The representation of the linear address appears in Figure 9.25. Because this addressing scheme can use PAE, virtual addresses are 48 bits in size but support 52-bit physical addresses (4,096 terabytes).

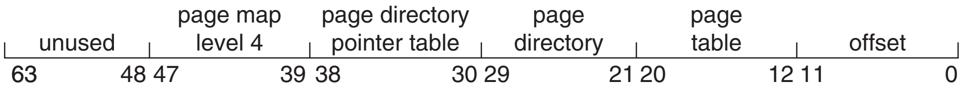

unused
page map
level 4
page directory
pointer table
page
directory
page
table
offset
63
48 47
39 38
30 29
21 20
12 11
0

**Figure 9.25** x86-64 linear address.

## 9.7 Example: ARMv8 Architecture

Although Intel chips have dominated the personal computer market for more than 30 years, chips for mobile devices such as smartphones and tablet computers often instead run on ARM processors. Interestingly, whereas Intel both designs and manufactures chips, ARM only designs them. It then licenses its architectural designs to chip manufacturers. Apple has licensed the ARM design for its iPhone and iPad mobile devices, and most Android-based smartphones use ARM processors as well. In addition to mobile devices, ARM also provides architecture designs for real-time embedded systems. Because of the abundance of devices that run on the ARM architecture, over 100 billion ARM processors have been produced, making it the most widely used architecture when measured in the quantity of chips produced. In this section, we describe the 64-bit ARMv8 architecture.

The ARMv8 has three different **translation granules**: 4 KB, 16 KB, and 64 KB. Each translation granule provides different page sizes, as well as larger sections of contiguous memory, known as **regions**. The page and region sizes for the different translation granules are shown below:

| Translation Granule Size | Page Size | Region Size |
|---|---|---|
| 4 KB | 4 KB | 2 MB, 1 GB |
| 16 KB | 16 KB | 32 MB |
| 64 KB | 64 KB | 512 MB |

For 4-KB and 16-KB granules, up to four levels of paging may be used, with up to three levels of paging for 64-KB granules. Figure 9.26 illustrates the ARMv8 address structure for the 4-KB translation granule with up to four levels of paging. (Notice that although ARMv8 is a 64-bit architecture, only 48 bits are currently used.) The four-level hierarchical paging structure for the 4-KB translation granule is illustrated in Figure 9.27. (The TTBR register is the **translation table base register** and points to the level 0 table for the current thread.)

If all four levels are used, the offset (bits 0–11 in Figure 9.26) refers to the offset within a 4-KB page. However, notice that the table entries for level 1 and

### *64-BIT COMPUTING*

History has taught us that even though memory capacities, CPU speeds, and similar computer capabilities seem large enough to satisfy demand for the foreseeable future, the growth of technology ultimately absorbs available capacities, and we find ourselves in need of additional memory or processing power, often sooner than we think. What might the future of technology bring that would make a 64-bit address space seem too small?

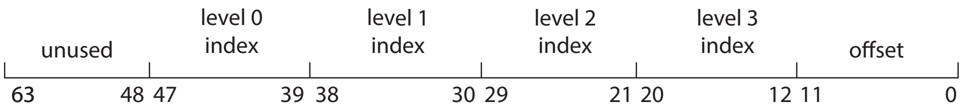

unused
level 0 index
level 1 index
level 2 index
level 3 index
offset
63
48 47
39 38
30 29
21 20
12 11
0

**Figure 9.26** ARM 4-KB translation granule.

level 2 may refer either to another table or to a 1-GB region (level-1 table) or 2-MB region (level-2 table). As an example, if the level-1 table refers to a 1-GB region rather than a level-2 table, the low-order 30 bits (bits 0–29 in Figure 9.26) are used as an offset into this 1-GB region. Similarly, if the level-2 table refers to a 2-MB region rather than a level-3 table, the low-order 21 bits (bits 0–20 in Figure 9.26) refer to the offset within this 2-MB region.

The ARM architecture also supports two levels of TLBs. At the inner level are two **micro TLBs**—a TLB for data and another for instructions. The micro TLB supports ASIDs as well. At the outer level is a single **main TLB**. Address translation begins at the micro-TLB level. In the case of a miss, the main TLB is then checked. If both TLBs yield misses, a page table walk must be performed in hardware.

## 9.8 Summary

- Memory is central to the operation of a modern computer system and consists of a large array of bytes, each with its own address.
- One way to allocate an address space to each process is through the use of base and limit registers. The base register holds the smallest legal physical memory address, and the limit specifies the size of the range.

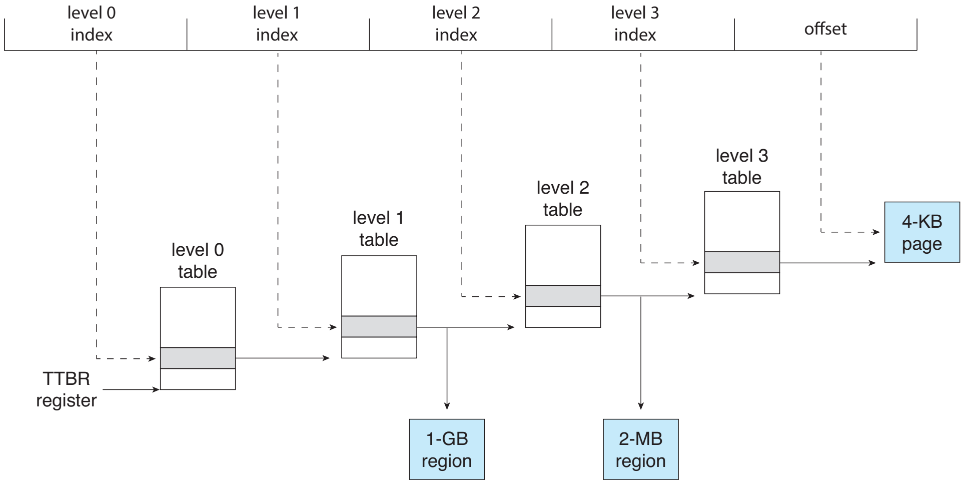

level 0 index
level 1 index
level 2 index
level 3 index
offset
level 3 table
level 2 table
4-KB page
level 1 table
level 0 table
TTBR register
1-GB region
2-MB region

**Figure 9.27** ARM four-level hierarchical paging.

- Binding symbolic address references to actual physical addresses may occur during (1) compile, (2) load, or (3) execution time.
- An address generated by the CPU is known as a logical address, which the memory management unit (MMU) translates to a physical address in memory.
- One approach to allocating memory is to allocate partitions of contiguous memory of varying sizes. These partitions may be allocated based on three possible strategies: (1) first fit, (2) best fit, and (3) worst fit.
- Modern operating systems use paging to manage memory. In this process, physical memory is divided into fixed-sized blocks called frames and logical memory into blocks of the same size called pages.
- When paging is used, a logical address is divided into two parts: a page number and a page offset. The page number serves as an index into a per-process page table that contains the frame in physical memory that holds the page. The offset is the specific location in the frame being referenced.
- A translation look-aside buffer (TLB) is a hardware cache of the page table. Each TLB entry contains a page number and its corresponding frame.
- Using a TLB in address translation for paging systems involves obtaining the page number from the logical address and checking if the frame for the page is in the TLB. If it is, the frame is obtained from the TLB. If the frame is not present in the TLB, it must be retrieved from the page table.
- Hierarchical paging involves dividing a logical address into multiple parts, each referring to different levels of page tables. As addresses expand beyond 32 bits, the number of hierarchical levels may become large. Two strategies that address this problem are hashed page tables and inverted page tables.
- Swapping allows the system to move pages belonging to a process to disk to increase the degree of multiprogramming.
- The Intel 32-bit architecture has two levels of page tables and supports either 4-KB or 4-MB page sizes. This architecture also supports page-address extension, which allows 32-bit processors to access a physical address space larger than 4 GB. The x86-64 and ARMv9 architectures are 64-bit architectures that use hierarchical paging.

## Practice Exercises

**9.1** Name two differences between logical and physical addresses.

**9.2** Why are page sizes always powers of 2?

**9.3** Consider a system in which a program can be separated into two parts: code and data. The CPU knows whether it wants an instruction (instruction fetch) or data (data fetch or store). Therefore, two base–limit register pairs are provided: one for instructions and one for data. The instruction base–limit register pair is automatically read-only, so programs can be shared among different users. Discuss the advantages and disadvantages of this scheme.

**9.4** Consider a logical address space of 64 pages of 1,024 words each, mapped onto a physical memory of 32 frames.

a. How many bits are there in the logical address?

b. How many bits are there in the physical address?

**9.5** What is the effect of allowing two entries in a page table to point to the same page frame in memory? Explain how this effect could be used to decrease the amount of time needed to copy a large amount of memory from one place to another. What effect would updating some byte on one page have on the other page?

**9.6** Given six memory partitions of 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 115 KB, 500 KB, 358 KB, 200 KB, and 375 KB (in order)?

**9.7** Assuming a 1-KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers):

a. 3085

b. 42095

c. 215201

d. 650000

e. 2000001

**9.8** The BTV operating system has a 21-bit virtual address, yet on certain embedded devices, it has only a 16-bit physical address. It also has a 2-KB page size. How many entries are there in each of the following?

a. A conventional, single-level page table

b. An inverted page table

What is the maximum amount of physical memory in the BTV operating system?

**9.9** Consider a logical address space of 256 pages with a 4-KB page size, mapped onto a physical memory of 64 frames.

a. How many bits are required in the logical address?

b. How many bits are required in the physical address?

**9.10** Consider a computer system with a 32-bit logical address and 4-KB page size. The system supports up to 512 MB of physical memory. How many entries are there in each of the following?

a. A conventional, single-level page table

b. An inverted page table

## Further Reading

The concept of paging can be credited to the designers of the Atlas system, which has been described by [Kilburn et al. (1961)] and by [Howarth et al. (1961)].

[Hennessy and Patterson (2012)] explain the hardware aspects of TLBs, caches, and MMUs. [Jacob and Mudge (2001)] describe techniques for managing the TLB. [Fang et al. (2001)] evaluate support for large pages.

PAE support for Windows systems.is discussed in http://msdn.microsoft.com/en-us/library/windows/hardware/gg487512.aspx An overview of the ARM architecture is provided in http://www.arm.com/products/processors/cortex-a/cortex-a9.php

## Bibliography

**[Fang et al. (2001)]** Z. Fang, L. Zhang, J. B. Carter, W. C. Hsieh, and S. A. McKee, “Reevaluating Online Superpage Promotion with Hardware Support”, *Proceedings of the International Symposium on High-Performance Computer Architecture*, Volume 50, Number 5 (2001).

**[Hennessy and Patterson (2012)]** J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach,* Fifth Edition, Morgan Kaufmann (2012).

**[Howarth et al. (1961)]** D. J. Howarth, R. B. Payne, and F. H. Sumner, “The Manchester University Atlas Operating System, Part II: User’s Description”, *Computer Journal*, Volume 4, Number 3 (1961), pages 226–229.

**[Jacob and Mudge (2001)]** B. Jacob and T. Mudge, “Uniprocessor Virtual Memory Without TLBs”, *IEEE Transactions on Computers*, Volume 50, Number 5 (2001).

**[Kilburn et al. (1961)]** T. Kilburn, D. J. Howarth, R. B. Payne, and F. H. Sumner, “The Manchester University Atlas Operating System, Part I: Internal Organization”, *Computer Journal*, Volume 4, Number 3 (1961), pages 222–225.

# Chapter 9 Exercises

**9.11** Explain the difference between internal and external fragmentation.

**9.12** Consider the following process for generating binaries. A compiler is used to generate the object code for individual modules, and a linker is used to combine multiple object modules into a single program binary. How does the linker change the binding of instructions and data to memory addresses? What information needs to be passed from the compiler to the linker to facilitate the memory-binding tasks of the linker?

**9.13** Given six memory partitions of 100 MB, 170 MB, 40 MB, 205 MB, 300 MB, and 185 MB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 200 MB, 15 MB, 185 MB, 75 MB, 175 MB, and 80 MB (in order)? Indicate which—if any—requests cannot be satisfied. Comment on how efficiently each of the algorithms manages memory.

**9.14** Most systems allow a program to allocate more memory to its address space during execution. Allocation of data in the heap segments of programs is an example of such allocated memory. What is required to support dynamic memory allocation in the following schemes?

a. Contiguous memory allocation

b. Paging

**9.15** Compare the memory organization schemes of contiguous memory allocation and paging with respect to the following issues:

a. External fragmentation

b. Internal fragmentation

c. Ability to share code across processes

**9.16** On a system with paging, a process cannot access memory that it does not own. Why? How could the operating system allow access to additional memory? Why should it or should it not?

**9.17** Explain why mobile operating systems such as iOS and Android do not support swapping.

**9.18** Although Android does not support swapping on its boot disk, it is possible to set up a swap space using a separate SD nonvolatile memory card. Why would Android disallow swapping on its boot disk yet allow it on a secondary disk?

**9.19** Explain why address-space identifiers (ASIDs) are used in TLBs.

**9.20** Program binaries in many systems are typically structured as follows. Code is stored starting with a small, fixed virtual address, such as 0. The code segment is followed by the data segment, which is used for storing the program variables. When the program starts executing, the stack is allocated at the other end of the virtual address space and is allowed to grow toward lower virtual addresses. What is the significance of this structure for the following schemes?

a. Contiguous memory allocation

b. Paging

**9.21** Assuming a 1-KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers)?

a. 21205

b. 164250

c. 121357

d. 16479315

e. 27253187

**9.22** The MPV operating system is designed for embedded systems and has a 24-bit virtual address, a 20-bit physical address, and a 4-KB page size. How many entries are there in each of the following?

a. A conventional, single-level page table

b. An inverted page table

What is the maximum amount of physical memory in the MPV operating system?

**9.23** Consider a logical address space of 2,048 pages with a 4-KB page size, mapped onto a physical memory of 512 frames.

a. How many bits are required in the logical address?

b. How many bits are required in the physical address?

**9.24** Consider a computer system with a 32-bit logical address and 8-KB page size. The system supports up to 1 GB of physical memory. How many entries are there in each of the following?

a. A conventional, single-level page table

b. An inverted page table

**9.25** Consider a paging system with the page table stored in memory.

a. If a memory reference takes 50 nanoseconds, how long does a paged memory reference take?

b. If we add TLBs, and if 75 percent of all page-table references are found in the TLBs, what is the effective memory reference time? (Assume that finding a page-table entry in the TLBs takes 2 nanoseconds, if the entry is present.)

**9.26** What is the purpose of paging the page tables?

**9.27** Consider the IA-32 address-translation scheme shown in Figure 9.22.

a. Describe all the steps taken by the IA-32 in translating a logical address into a physical address.

b. What are the advantages to the operating system of hardware that provides such complicated memory translation?

c. Are there any disadvantages to this address-translation system? If so, what are they? If not, why is this scheme not used by every manufacturer?

## Programming Problems

**9.28** Assume that a system has a 32-bit virtual address with a 4-KB page size. Write a C program that is passed a virtual address (in decimal) on the command line and have it output the page number and offset for the given address. As an example, your program would run as follows:

```
./addresses 19986
```

Your program would output:

```
The address 19986 contains:
page number = 4
offset = 3602
```

Writing this program will require using the appropriate data type to store 32 bits. We encourage you to use `unsigned` data types as well.

## Programming Projects

### Contiguous Memory Allocation

In Section 9.2, we presented different algorithms for contiguous memory allocation. This project will involve managing a contiguous region of memory of size *MAX* where addresses may range from 0 ... *MAX* – 1. Your program must respond to four different requests:

1. Request for a contiguous block of memory
2. Release of a contiguous block of memory
3. Compact unused holes of memory into one single block
4. Report the regions of free and allocated memory

Your program will be passed the initial amount of memory at startup. For example, the following initializes the program with 1 MB (1,048,576 bytes) of memory:

```
./allocator 1048576
```

Once your program has started, it will present the user with the following prompt:

```
allocator>
```

It will then respond to the following commands: `RQ` (request), `RL` (release), `C` (compact), `STAT` (status report), and `X` (exit).

A request for 40,000 bytes will appear as follows:

```
allocator>RQ P0 40000 W
```

The first parameter to the `RQ` command is the new process that requires the memory, followed by the amount of memory being requested, and finally the strategy. (In this situation, "`W`" refers to worst fit.)

Similarly, a release will appear as:

```
allocator>RL P0
```

This command will release the memory that has been allocated to process `P0`.

The command for compaction is entered as:

```
allocator>C
```

This command will compact unused holes of memory into one region.

Finally, the `STAT` command for reporting the status of memory is entered as:

```
allocator>STAT
```

Given this command, your program will report the regions of memory that are allocated and the regions that are unused. For example, one possible arrangement of memory allocation would be as follows:

```
Addresses [0:315000] Process P1
Addresses [315001: 512500] Process P3
Addresses [512501:625575] Unused
Addresses [625575:725100] Process P6
Addresses [725001] . . .
```

## Allocating Memory

Your program will allocate memory using one of the three approaches highlighted in Section 9.2.2, depending on the flag that is passed to the `RQ` command. The flags are:

- `F`—first fit
- `B`—best fit
- `W`—worst fit

This will require that your program keep track of the different holes representing available memory. When a request for memory arrives, it will allocate the memory from one of the available holes based on the allocation strategy. If there is insufficient memory to allocate to a request, it will output an error message and reject the request.

Your program will also need to keep track of which region of memory has been allocated to which process. This is necessary to support the `STAT` command and is also needed when memory is released via the `RL` command, as the process releasing memory is passed to this command. If a partition being released is adjacent to an existing hole, be sure to combine the two holes into a single hole.

## Compaction

If the user enters the `C` command, your program will compact the set of holes into one larger hole. For example, if you have four separate holes of size 550 KB, 375 KB, 1,900 KB, and 4,500 KB, your program will combine these four holes into one large hole of size 7,325 KB.

There are several strategies for implementing compaction, one of which is suggested in Section 9.2.3. Be sure to update the beginning address of any processes that have been affected by compaction.

# CHAPTER 10

# Virtual Memory

In Chapter 9, we discussed various memory-management strategies used in computer systems. All these strategies have the same goal: to keep many processes in memory simultaneously to allow multiprogramming. However, they tend to require that an entire process be in memory before it can execute.

Virtual memory is a technique that allows the execution of processes that are not completely in memory. One major advantage of this scheme is that programs can be larger than physical memory. Further, virtual memory abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the programmer from physical memory. This technique frees programmers from the concerns of memory-storage limitations. Virtual memory also allows processes to share files and libraries, and to implement shared memory. In addition, it provides an efficient mechanism for process creation. Virtual memory is not easy to implement, however, and may substantially decrease performance if it is used carelessly. In this chapter, we provide a detailed overview of virtual memory, examine how it is implemented, and explore its complexity and benefits.

## CHAPTER OBJECTIVES

- Define virtual memory and describe its benefits.
- Illustrate how pages are loaded into memory using demand paging.
- Apply the FIFO, optimal, and LRU page-replacement algorithms.
- Describe the working set of a process, and explain how it is related to program locality.
- Describe how Linux, Windows 10, and Solaris manage virtual memory.
- Design a virtual memory manager simulation in the C programming language.

## 10.1 Background

The memory-management algorithms outlined in Chapter 9 are necessary because of one basic requirement: the instructions being executed must be in physical memory. The first approach to meeting this requirement is to place the entire logical address space in physical memory. Dynamic linking can help to ease this restriction, but it generally requires special precautions and extra work by the programmer.

The requirement that instructions must be in physical memory to be executed seems both necessary and reasonable; but it is also unfortunate, since it limits the size of a program to the size of physical memory. In fact, an examination of real programs shows us that, in many cases, the entire program is not needed. For instance, consider the following:

- Programs often have code to handle unusual error conditions. Since these errors seldom, if ever, occur in practice, this code is almost never executed.
- Arrays, lists, and tables are often allocated more memory than they actually need. An array may be declared 100 by 100 elements, even though it is seldom larger than 10 by 10 elements.
- Certain options and features of a program may be used rarely. For instance, the routines on U.S. government computers that balance the budget have not been used in many years.

Even in those cases where the entire program is needed, it may not all be needed at the same time.

The ability to execute a program that is only partially in memory would confer many benefits:

- A program would no longer be constrained by the amount of physical memory that is available. Users would be able to write programs for an extremely large ***virtual*** address space, simplifying the programming task.
- Because each program could take less physical memory, more programs could be run at the same time, with a corresponding increase in CPU utilization and throughput but with no increase in response time or turnaround time.
- Less I/O would be needed to load or swap portions of programs into memory, so each program would run faster.

Thus, running a program that is not entirely in memory would benefit both the system and its users.

**Virtual memory** involves the separation of logical memory as perceived by developers from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available (Figure 10.1). Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the amount of physical memory available; she can concentrate instead on programming the problem that is to be solved.

The **virtual address space** of a process refers to the logical (or virtual) view of how a process is stored in memory. Typically, this view is that a process begins at a certain logical address—say, address 0—and exists in contiguous memory, as shown in Figure 10.2. Recall from Chapter 9, though, that in fact physical memory is organized in page frames and that the physical page frames assigned to a process may not be contiguous. It is up to the memory-

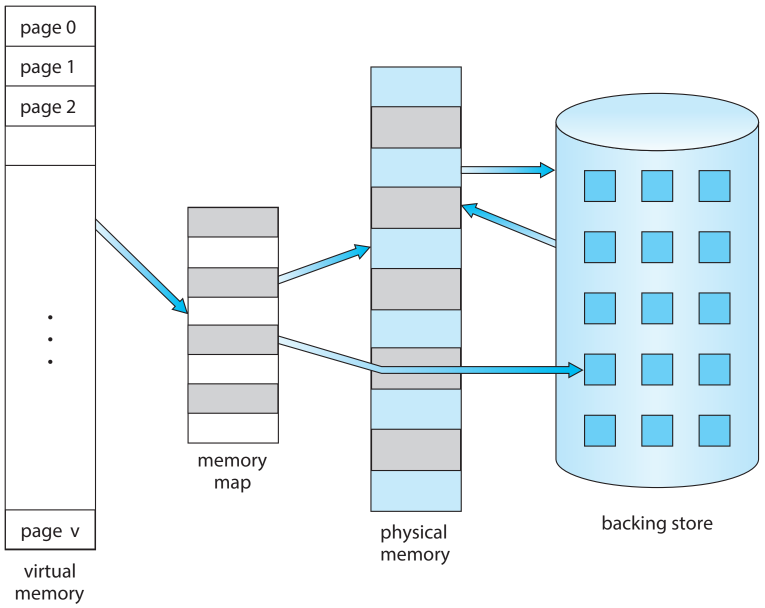

page 0
page 1
page 2
page v
virtual memory
memory map
physical memory
backing store

**Figure 10.1** Diagram showing virtual memory that is larger than physical memory.

management unit (MMU) to map logical pages to physical page frames in memory.

Note in Figure 10.2 that we allow the heap to grow upward in memory as it is used for dynamic memory allocation. Similarly, we allow for the stack to grow downward in memory through successive function calls. The large blank space (or hole) between the heap and the stack is part of the virtual address space but will require actual physical pages only if the heap or stack grows. Virtual address spaces that include holes are known as **sparse** address spaces. Using a sparse address space is beneficial because the holes can be filled as the stack or heap segments grow or if we wish to dynamically link libraries (or possibly other shared objects) during program execution.

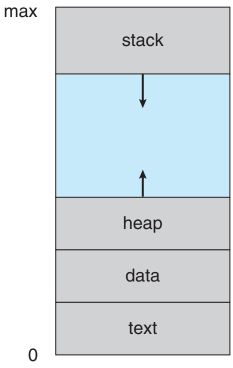

max
stack
heap
data
text
0

**Figure 10.2** Virtual address space of a process in memory.

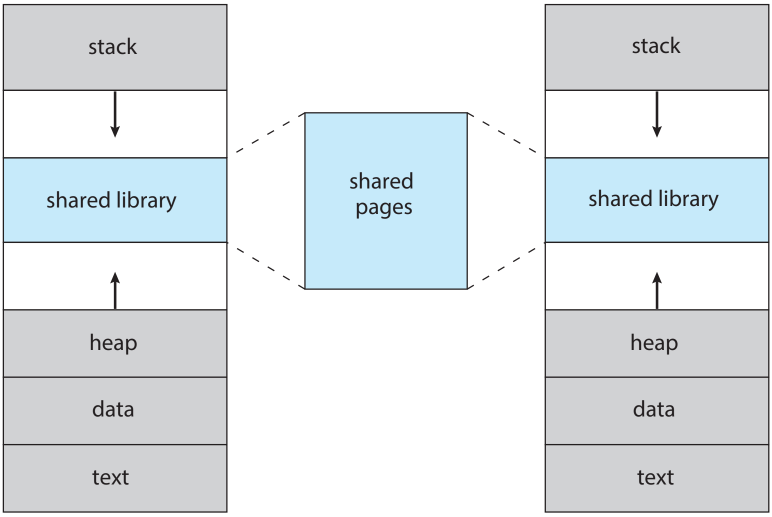


**Figure 10.3** Shared library using virtual memory.

In addition to separating logical memory from physical memory, virtual memory allows files and memory to be shared by two or more processes through page sharing (Section 9.3.4). This leads to the following benefits:

- System libraries such as the standard C library can be shared by several processes through mapping of the shared object into a virtual address space. Although each process considers the libraries to be part of its virtual address space, the actual pages where the libraries reside in physical memory are shared by all the processes (Figure 10.3). Typically, a library is mapped read-only into the space of each process that is linked with it.
- Similarly, processes can share memory. Recall from Chapter 3 that two or more processes can communicate through the use of shared memory. Virtual memory allows one process to create a region of memory that it can share with another process. Processes sharing this region consider it part of their virtual address space, yet the actual physical pages of memory are shared, much as is illustrated in Figure 10.3.
- Pages can be shared during process creation with the `fork()` system call, thus speeding up process creation.

We further explore these—and other—benefits of virtual memory later in this chapter. First, though, we discuss implementing virtual memory through demand paging.

## 10.2 Demand Paging

Consider how an executable program might be loaded from secondary storage into memory. One option is to load the entire program in physical memory at program execution time. However, a problem with this approach is that we may not initially ***need*** the entire program in memory. Suppose a program starts with a list of available options from which the user is to select. Loading the entire program into memory results in loading the executable code for ***all*** options, regardless of whether or not an option is ultimately selected by the user.

An alternative strategy is to load pages only as they are needed. This technique is known as **demand paging** and is commonly used in virtual memory systems. With demand-paged virtual memory, pages are loaded only when they are ***demanded*** during program execution. Pages that are never accessed are thus never loaded into physical memory. A demand-paging system is similar to a paging system with swapping (Section 9.5.2) where processes reside in secondary memory (usually an HDD or NVM device). Demand paging explains one of the primary benefits of virtual memory—by loading only the portions of programs that are needed, memory is used more efficiently.

### 10.2.1 Basic Concepts

The general concept behind demand paging, as mentioned, is to load a page in memory only when it is needed. As a result, while a process is executing, some pages will be in memory, and some will be in secondary storage. Thus, we need some form of hardware support to distinguish between the two. The valid–invalid bit scheme described in Section 9.3.3 can be used for this purpose. This

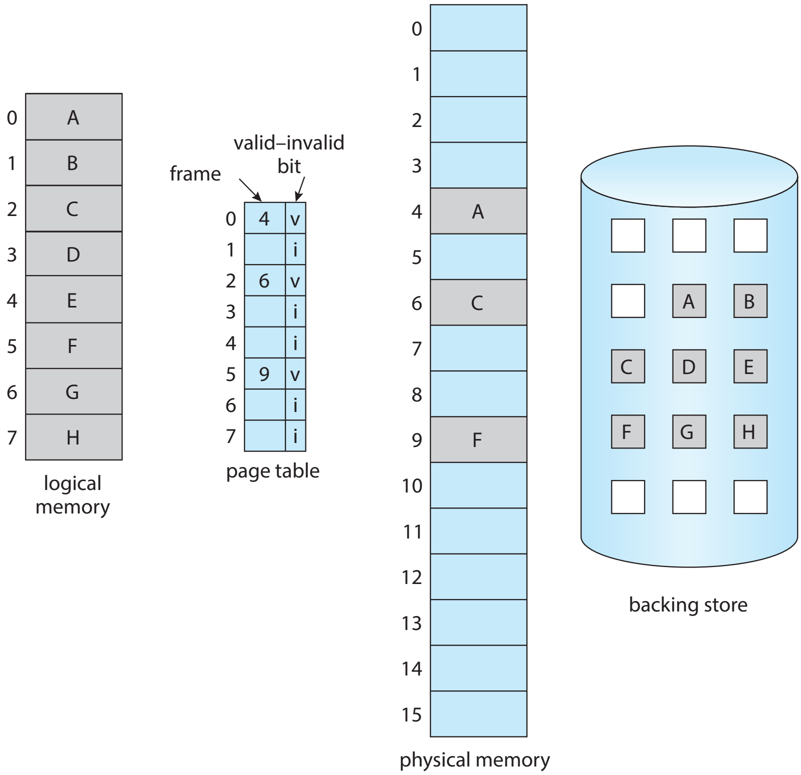

0
1
2
3
4
5
6
7
A
B
C
D
E
F
G
H
logical memory
frame
valid–invalid bit
0 4 v
1 i
2 6 v
3 i
4 i
5 9 v
6 i
7 i
page table
0
1
2
3
4 A
5
6 C
7
8
9 F
10
11
12
13
14
15
physical memory
A
B
C
D
E
F
G
H
backing store

**Figure 10.4** Page table when some pages are not in main memory.

time, however, when the bit is set to “valid,” the associated page is both legal and in memory. If the bit is set to “invalid,” the page either is not valid (that is, not in the logical address space of the process) or is valid but is currently in secondary storage. The page-table entry for a page that is brought into memory is set as usual, but the page-table entry for a page that is not currently in memory is simply marked invalid. This situation is depicted in Figure 10.4. (Notice that marking a page invalid will have no effect if the process never attempts to access that page.)

But what happens if the process tries to access a page that was not brought into memory? Access to a page marked invalid causes a **page fault**. The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the operating system. This trap is the result of the operating system's failure to bring the desired page into memory. The procedure for handling this page fault is straightforward (Figure 10.5):

1. We check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access.
2. If the reference was invalid, we terminate the process. If it was valid but we have not yet brought in that page, we now page it in.
3. We find a free frame (by taking one from the free-frame list, for example).

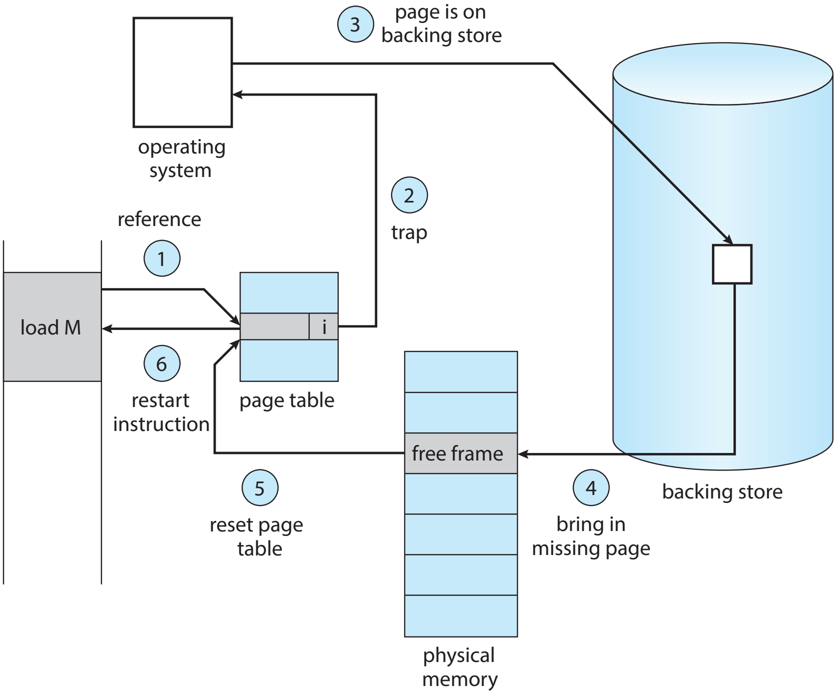

③ page is on backing store
operating system
② trap
reference
①
load M
i
⑥ restart instruction
page table
free frame
⑤ reset page table
④ bring in missing page
backing store
physical memory

**Figure 10.5** Steps in handling a page fault.

4. We schedule a secondary storage operation to read the desired page into the newly allocated frame.
5. When the storage read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.

In the extreme case, we can start executing a process with ***no*** pages in memory. When the operating system sets the instruction pointer to the first instruction of the process, which is on a non-memory-resident page, the process immediately faults for the page. After this page is brought into memory, the process continues to execute, faulting as necessary until every page that it needs is in memory. At that point, it can execute with no more faults. This scheme is **pure demand paging**: never bring a page into memory until it is required.

Theoretically, some programs could access several new pages of memory with each instruction execution (one page for the instruction and many for data), possibly causing multiple page faults per instruction. This situation would result in unacceptable system performance. Fortunately, analysis of running processes shows that this behavior is exceedingly unlikely. Programs tend to have **locality of reference**, described in Section 10.6.1, which results in reasonable performance from demand paging.

The hardware to support demand paging is the same as the hardware for paging and swapping:

- **Page table**. This table has the ability to mark an entry invalid through a valid–invalid bit or a special value of protection bits.
- **Secondary memory**. This memory holds those pages that are not present in main memory. The secondary memory is usually a high-speed disk or NVM device. It is known as the swap device, and the section of storage used for this purpose is known as **swap space**. Swap-space allocation is discussed in Chapter 11.

A crucial requirement for demand paging is the ability to restart any instruction after a page fault. Because we save the state (registers, condition code, instruction counter) of the interrupted process when the page fault occurs, we must be able to restart the process in ***exactly*** the same place and state, except that the desired page is now in memory and is accessible. In most cases, this requirement is easy to meet. A page fault may occur at any memory reference. If the page fault occurs on the instruction fetch, we can restart by fetching the instruction again. If a page fault occurs while we are fetching an operand, we must fetch and decode the instruction again and then fetch the operand.

As a worst-case example, consider a three-address instruction such as ADD the content of A to B, placing the result in C. These are the steps to execute this instruction:

1. Fetch and decode the instruction (ADD).
2. Fetch A.

3. Fetch B.
4. Add A and B.
5. Store the sum in C.

If we fault when we try to store in C (because C is in a page not currently in memory), we will have to get the desired page, bring it in, correct the page table, and restart the instruction. The restart will require fetching the instruction again, decoding it again, fetching the two operands again, and then adding again. However, there is not much repeated work (less than one complete instruction), and the repetition is necessary only when a page fault occurs.

The major difficulty arises when one instruction may modify several different locations. For example, consider the IBM System 360/370 MVC (move character) instruction, which can move up to 256 bytes from one location to another (possibly overlapping) location. If either block (source or destination) straddles a page boundary, a page fault might occur after the move is partially done. In addition, if the source and destination blocks overlap, the source block may have been modified, in which case we cannot simply restart the instruction.

This problem can be solved in two different ways. In one solution, the microcode computes and attempts to access both ends of both blocks. If a page fault is going to occur, it will happen at this step, before anything is modified. The move can then take place; we know that no page fault can occur, since all the relevant pages are in memory. The other solution uses temporary registers to hold the values of overwritten locations. If there is a page fault, all the old values are written back into memory before the trap occurs. This action restores memory to its state before the instruction was started, so that the instruction can be repeated.

This is by no means the only architectural problem resulting from adding paging to an existing architecture to allow demand paging, but it illustrates some of the difficulties involved. Paging is added between the CPU and the memory in a computer system. It should be entirely transparent to a process. Thus, people often assume that paging can be added to any system. Although this assumption is true for a non-demand-paging environment, where a page fault represents a fatal error, it is not true where a page fault means only that an additional page must be brought into memory and the process restarted.

### 10.2.2 Free-Frame List

When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory. To resolve page faults, most operating systems maintain a **free-frame list**, a pool of free frames for satisfying such requests (Figure 10.6). (Free frames must also be allocated when the stack or heap segments from a process expand.) Operating systems typically allo-

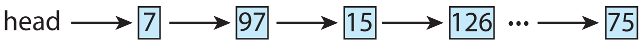


**Figure 10.6** List of free frames.

cate free frames using a technique known as **zero-fill-on-deman** . Zero-fill-on-demand frames are "zeroed-out" before being allocated, thus erasing their previous contents. (Consider the potential security implications of ***not*** clearing out the contents of a frame before reassigning it.)

When a system starts up, all available memory is placed on the free-frame list. As free frames are requested (for example, through demand paging), the size of the free-frame list shrinks. At some point, the list either falls to zero or falls below a certain threshold, at which point it must be repopulated. We cover strategies for both of these situations in Section 10.4.

### 10.2.3 Performance of Demand Paging

Demand paging can significantly affect the performance of a computer system. To see why, let's compute the **effective access time** for a demand-paged memory. Assume the memory-access time, denoted *ma,* is 10 nanoseconds. As long as we have no page faults, the effective access time is equal to the memory access time. If, however, a page fault occurs, we must first read the relevant page from secondary storage and then access the desired word.

Let $p$ be the probability of a page fault ($0 \le p \le 1$). We would expect $p$ to be close to zero—that is, we would expect to have only a few page faults. The effective access time is then

$$\text{effective access time} = (1 - p) \times ma + p \times \text{page fault time}.$$

To compute the effective access time, we must know how much time is needed to service a page fault. A page fault causes the following sequence to occur:

1. Trap to the operating system.
2. Save the registers and process state.
3. Determine that the interrupt was a page fault.
4. Check that the page reference was legal, and determine the location of the page in secondary storage.
5. Issue a read from the storage to a free frame:
    a. Wait in a queue until the read request is serviced.
    b. Wait for the device seek and/or latency time.
    c. Begin the transfer of the page to a free frame.
6. While waiting, allocate the CPU core to some other process.
7. Receive an interrupt from the storage I/O subsystem (I/O completed).
8. Save the registers and process state for the other process (if step 6 is executed).
9. Determine that the interrupt was from the secondary storage device.
10. Correct the page table and other tables to show that the desired page is now in memory.
11. Wait for the CPU core to be allocated to this process again.

12. Restore the registers, process state, and new page table, and then resume the interrupted instruction.

Not all of these steps are necessary in every case. For example, we are assuming that, in step 6, the CPU is allocated to another process while the I/O occurs. This arrangement allows multiprogramming to maintain CPU utilization but requires additional time to resume the page-fault service routine when the I/O transfer is complete.

In any case, there are three major task components of the page-fault service time:

1. Service the page-fault interrupt.
2. Read in the page.
3. Restart the process.

The first and third tasks can be reduced, with careful coding, to several hundred instructions. These tasks may take from 1 to 100 microseconds each. Let's consider the case of HDDs being used as the paging device. The page-switch time will probably be close to 8 milliseconds. (A typical hard disk has an average latency of 3 milliseconds, a seek of 5 milliseconds, and a transfer time of 0.05 milliseconds. Thus, the total paging time is about 8 milliseconds, including hardware and software time.) Remember also that we are looking at only the device-service time. If a queue of processes is waiting for the device, we have to add queuing time as we wait for the paging device to be free to service our request, increasing even more the time to page in.

With an average page-fault service time of 8 milliseconds and a memory-access time of 200 nanoseconds, the effective access time in nanoseconds is

$$\begin{aligned}\text{effective access time} &= (1 - p) \times (200) + p\ (8\ \text{milliseconds})\\ &= (1 - p) \times 200 + p \times 8{,}000{,}000\\ &= 200 + 7{,}999{,}800 \times p.\end{aligned}$$

We see, then, that the effective access time is directly proportional to the **page-fault rate**. If one access out of 1,000 causes a page fault, the effective access time is 8.2 microseconds. The computer will be slowed down by a factor of 40 because of demand paging! If we want performance degradation to be less than 10 percent, we need to keep the probability of page faults at the following level:

$$\begin{aligned}&220 > 200 + 7{,}999{,}800 \times p,\\ &20 > 7{,}999{,}800 \times p,\\ &p < 0.0000025.\end{aligned}$$

That is, to keep the slowdown due to paging at a reasonable level, we can allow fewer that one memory access out of 399,990 to page-fault. In sum, it is important to keep the page-fault rate low in a demand-paging system. Otherwise, the effective access time increases, slowing process execution dramatically.

An additional aspect of demand paging is the handling and overall use of swap space. I/O to swap space is generally faster than that to the file system. It is faster because swap space is allocated in much larger blocks, and file lookups and indirect allocation methods are not used (Chapter 11). One option for the system to gain better paging throughput is by copying an entire file image into the swap space at process startup and then performing demand paging from the swap space. The obvious disadvantage of this approach is the copying of the file image at program start-up. A second option—and one practiced by several operating systems, including Linux and Windows—is to demand-page from the file system initially but to write the pages to swap space as they are replaced. This approach will ensure that only needed pages are read from the file system but that all subsequent paging is done from swap space.

Some systems attempt to limit the amount of swap space used through demand paging of binary executable files. Demand pages for such files are brought directly from the file system. However, when page replacement is called for, these frames can simply be overwritten (because they are never modified), and the pages can be read in from the file system again if needed. Using this approach, the file system itself serves as the backing store. However, swap space must still be used for pages not associated with a file (known as **anonymous memory**); these pages include the stack and heap for a process. This method appears to be a good compromise and is used in several systems, including Linux and BSD UNIX.

As described in Section 9.5.3, mobile operating systems typically do not support swapping. Instead, these systems demand-page from the file system and reclaim read-only pages (such as code) from applications if memory becomes constrained. Such data can be demand-paged from the file system if it is later needed. Under iOS, anonymous memory pages are never reclaimed from an application unless the application is terminated or explicitly releases the memory. In Section 10.7, we cover compressed memory, a commonly used alternative to swapping in mobile systems.

## 10.3 Copy-on-Write

In Section 10.2, we illustrated how a process can start quickly by demand-paging in the page containing the first instruction. However, process creation using the `fork()` system call may initially bypass the need for demand paging by using a technique similar to page sharing (covered in Section 9.3.4). This technique provides rapid process creation and minimizes the number of new pages that must be allocated to the newly created process.

Recall that the `fork()` system call creates a child process that is a duplicate of its parent. Traditionally, `fork()` worked by creating a copy of the parent's address space for the child, duplicating the pages belonging to the parent. However, considering that many child processes invoke the `exec()` system call immediately after creation, the copying of the parent's address space may be unnecessary. Instead, we can use a technique known as **copy-on-write**, which works by allowing the parent and child processes initially to share the same pages. These shared pages are marked as copy-on-write pages, meaning that if either process writes to a shared page, a copy of the shared page is created. Copy-on-write is illustrated in Figures 10.7 and 10.8, which show the contents of the physical memory before and after process 1 modifies page C.

For example, assume that the child process attempts to modify a page containing portions of the stack, with the pages set to be copy-on-write. The operating system will obtain a frame from the free-frame list and create a copy

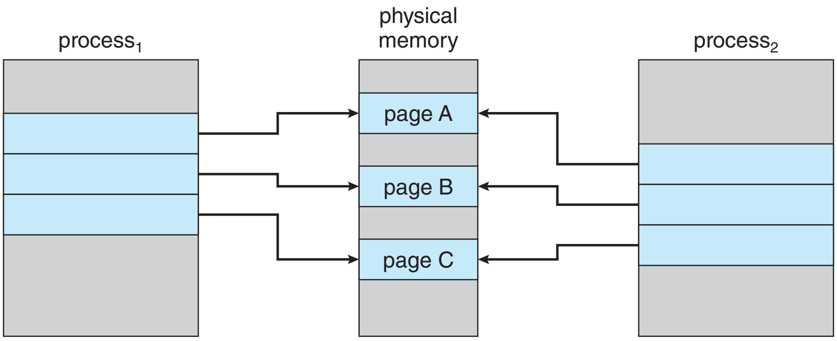


**Figure 10.7** Before process 1 modifies page C.

of this page, mapping it to the address space of the child process. The child process will then modify its copied page and not the page belonging to the parent process. Obviously, when the copy-on-write technique is used, only the pages that are modified by either process are copied; all unmodified pages can be shared by the parent and child processes. Note, too, that only pages that can be modified need be marked as copy-on-write. Pages that cannot be modified (pages containing executable code) can be shared by the parent and child. Copy-on-write is a common technique used by several operating systems, including Windows, Linux, and macOS.

Several versions of UNIX (including Linux, macOS, and BSD UNIX) provide a variation of the `fork()` system call—`vfork()` (for **virtual memory fork**)—that operates differently from `fork()` with copy-on-write. With `vfork()`, the parent process is suspended, and the child process uses the address space of the parent. Because `vfork()` does not use copy-on-write, if the child process changes any pages of the parent's address space, the altered pages will be visible to the parent once it resumes. Therefore, `vfork()` must be used with caution to ensure that the child process does not modify the address space of the parent. `vfork()` is intended to be used when the child process calls `exec()` immediately after creation. Because no copying of pages takes place, `vfork()`

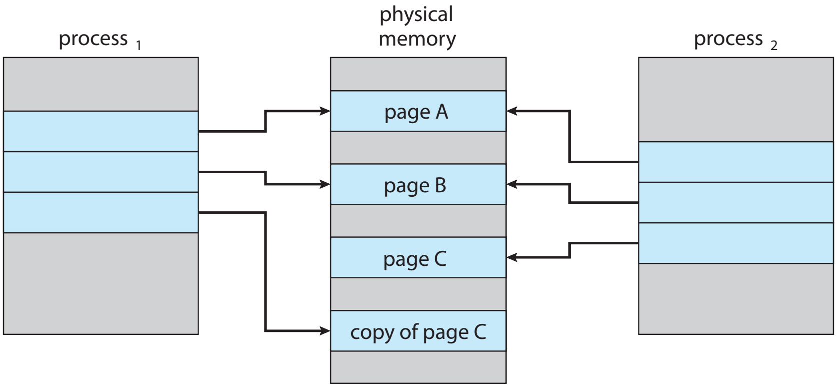


**Figure 10.8** After process 1 modifies page C.

is an extremely efficient method of process creation and is sometimes used to implement UNIX command-line shell interfaces.

## 10.4 Page Replacement

In our earlier discussion of the page-fault rate, we assumed that each page faults at most once, when it is first referenced. This representation is not strictly accurate, however. If a process of ten pages actually uses only half of them, then demand paging saves the I/O necessary to load the five pages that are never used. We could also increase our degree of multiprogramming by running twice as many processes. Thus, if we had forty frames, we could run eight processes, rather than the four that could run if each required ten frames (five of which were never used).

If we increase our degree of multiprogramming, we are **over-allocating** memory. If we run six processes, each of which is ten pages in size but actually uses only five pages, we have higher CPU utilization and throughput, with ten frames to spare. It is possible, however, that each of these processes, for a particular data set, may suddenly try to use all ten of its pages, resulting in a need for sixty frames when only forty are available.

Further, consider that system memory is not used only for holding program pages. Buffers for I/O also consume a considerable amount of memory. This use can increase the strain on memory-placement algorithms. Deciding how much memory to allocate to I/O and how much to program pages is a significant challenge. Some systems allocate a fixed percentage of memory for I/O buffers, whereas others allow both processes and the I/O subsystem to compete for all system memory. Section 14.6 discusses the integrated relationship between I/O buffers and virtual memory techniques.

Over-allocation of memory manifests itself as follows. While a process is executing, a page fault occurs. The operating system determines where the desired page is residing on secondary storage but then finds that there are ***no*** free frames on the free-frame list; all memory is in use. This situation is illustrated in Figure 10.9, where the fact that there are no free frames is depicted by a question mark.

The operating system has several options at this point. It could terminate the process. However, demand paging is the operating system's attempt to improve the computer system's utilization and throughput. Users should not be aware that their processes are running on a paged system—paging should be logically transparent to the user. So this option is not the best choice.

The operating system could instead use standard swapping and swap out a process, freeing all its frames and reducing the level of multiprogramming. However, as discussed in Section 9.5, standard swapping is no longer used by most operating systems due to the overhead of copying entire processes between memory and swap space. Most operating systems now combine swapping pages with **page replacement**, a technique we describe in detail in the remainder of this section.

### 10.4.1 Basic Page Replacement

Page replacement takes the following approach. If no frame is free, we find one that is not currently being used and free it. We can free a frame by writing its

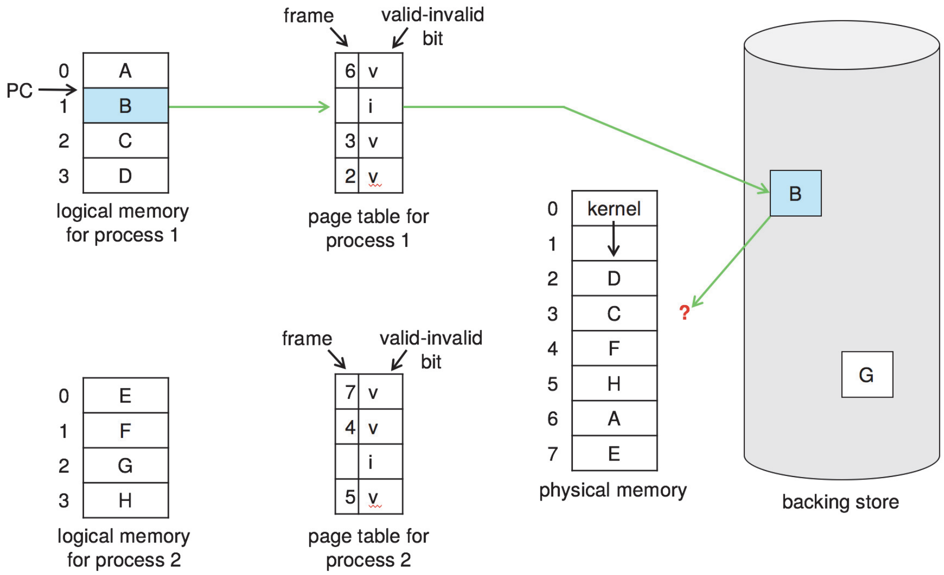

frame
valid-invalid bit
PC
0 A
1 B
2 C
3 D
logical memory for process 1
6 v
i
3 v
2 v
page table for process 1
0 kernel
1
2 D
3 C
4 F
5 H
6 A
7 E
physical memory
?
B
G
backing store
0 E
1 F
2 G
3 H
logical memory for process 2
7 v
4 v
i
5 v
page table for process 2

**Figure 10.9** Need for page replacement.

contents to swap space and changing the page table (and all other tables) to indicate that the page is no longer in memory (Figure 10.10). We can now use the freed frame to hold the page for which the process faulted. We modify the page-fault service routine to include page replacement:

1. Find the location of the desired page on secondary storage.
2. Find a free frame:
    a. If there is a free frame, use it.
    b. If there is no free frame, use a page-replacement algorithm to select a **victim frame**.
    c. Write the victim frame to secondary storage (if necessary); change the page and frame tables accordingly.
3. Read the desired page into the newly freed frame; change the page and frame tables.
4. Continue the process from where the page fault occurred.

Notice that, if no frames are free, *two* page transfers (one for the page-out and one for the page-in) are required. This situation effectively doubles the page-fault service time and increases the effective access time accordingly.

We can reduce this overhead by using a **modify bit** (or **dirty bit**). When this scheme is used, each page or frame has a modify bit associated with it in the hardware. The modify bit for a page is set by the hardware whenever any byte in the page is written into, indicating that the page has been modified. When we select a page for replacement, we examine its modify bit. If the bit is set,

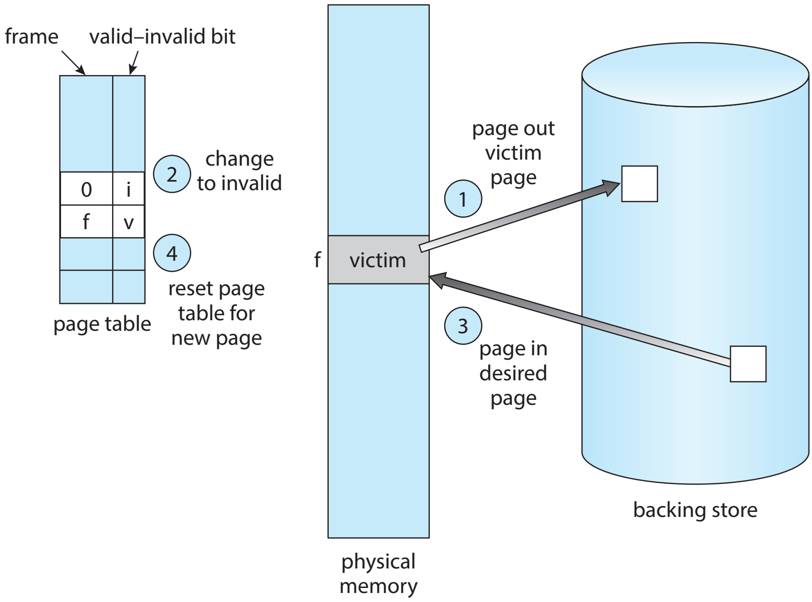


**Figure 10.10** Page replacement.

we know that the page has been modified since it was read in from secondary storage. In this case, we must write the page to storage. If the modify bit is not set, however, the page has ***not*** been modified since it was read into memory. In this case, we need not write the memory page to storage: it is already there. This technique also applies to read-only pages (for example, pages of binary code). Such pages cannot be modified; thus, they may be discarded when desired. This scheme can significantly reduce the time required to service a page fault, since it reduces I/O time by one-half ***if*** the page has not been modified.

Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory. With this mechanism, an enormous virtual memory can be provided for programmers on a smaller physical memory. With no demand paging, logical addresses are mapped into physical addresses, and the two sets of addresses can be different. All the pages of a process still must be in physical memory, however. With demand paging, the size of the logical address space is no longer constrained by physical memory. If we have a process of twenty pages, we can execute it in ten frames simply by using demand paging and using a replacement algorithm to find a free frame whenever necessary. If a page that has been modified is to be replaced, its contents are copied to secondary storage. A later reference to that page will cause a page fault. At that time, the page will be brought back into memory, perhaps replacing some other page in the process.

We must solve two major problems to implement demand paging: we must develop a **frame-allocation algorithm** and a **page-replacement algorithm**. That is, if we have multiple processes in memory, we must decide how many frames to allocate to each process; and when page replacement is required, we must select the frames that are to be replaced. Designing appropriate algorithms to solve these problems is an important task, because secondary storage

I/O is so expensive. Even slight improvements in demand-paging methods yield large gains in system performance.

There are many different page-replacement algorithms. Every operating system probably has its own replacement scheme. How do we select a particular replacement algorithm? In general, we want the one with the lowest page-fault rate.

We evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults. The string of memory references is called a **reference string**. We can generate reference strings artificially (by using a random-number generator, for example), or we can trace a given system and record the address of each memory reference. The latter choice produces a large number of data (on the order of 1 million addresses per second). To reduce the number of data, we use two facts.

First, for a given page size (and the page size is generally fixed by the hardware or system), we need to consider only the page number, rather than the entire address. Second, if we have a reference to a page *p*, then any references to page *p* that ***immediately*** follow will never cause a page fault. Page *p* will be in memory after the first reference, so the immediately following references will not fault.

For example, if we trace a particular process, we might record the following address sequence:

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103,
0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105

At 100 bytes per page, this sequence is reduced to the following reference string:

1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

To determine the number of page faults for a particular reference string and page-replacement algorithm, we also need to know the number of page frames available. Obviously, as the number of frames available increases, the number of page faults decreases. For the reference string considered previously, for example, if we had three or more frames, we would have only three faults—one fault for the first reference to each page. In contrast, with only one frame available, we would have a replacement with every reference, resulting in eleven faults. In general, we expect a curve such as that in Figure 10.11. As the number of frames increases, the number of page faults drops to some minimal level. Of course, adding physical memory increases the number of frames.

We next illustrate several page-replacement algorithms. In doing so, we use the reference string

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

for a memory with three frames.

### 10.4.2 FIFO Page Replacement

The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. Notice that it is not strictly necessary to record the time when a

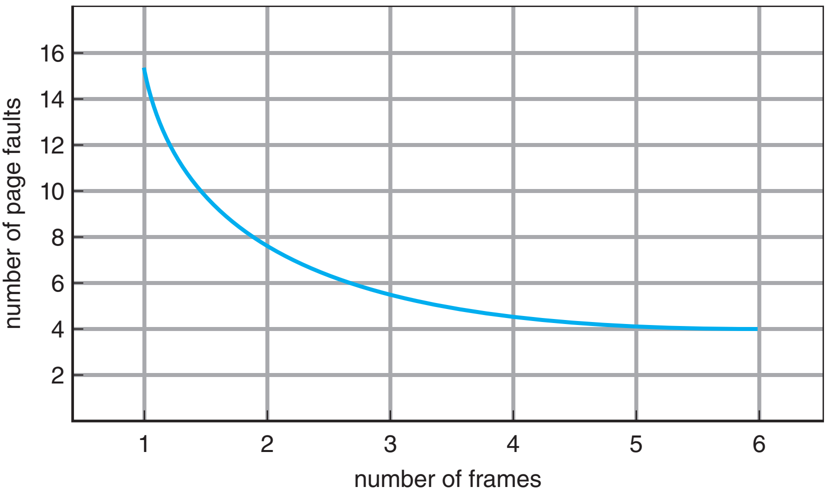


**Figure 10.11** Graph of page faults versus number of frames.

page is brought in. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.

For our example reference string, our three frames are initially empty. The first three references (7, 0, 1) cause page faults and are brought into these empty frames. The next reference (2) replaces page 7, because page 7 was brought in first. Since 0 is the next reference and 0 is already in memory, we have no fault for this reference. The first reference to 3 results in replacement of page 0, since it is now first in line. Because of this replacement, the next reference, to 0, will fault. Page 1 is then replaced by page 0. This process continues as shown in Figure 10.12. Every time a fault occurs, we show which pages are in our three frames. There are fifteen faults altogether.

The FIFO page-replacement algorithm is easy to understand and program. However, its performance is not always good. On the one hand, the page replaced may be an initialization module that was used a long time ago and is no longer needed. On the other hand, it could contain a heavily used variable that was initialized early and is in constant use.

Notice that, even if we select for replacement a page that is in active use, everything still works correctly. After we replace an active page with a new

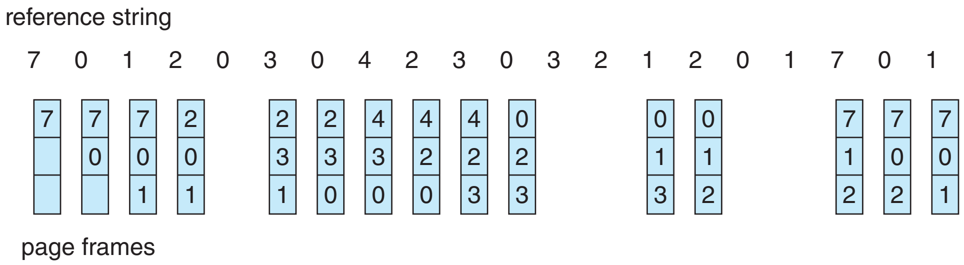


**Figure 10.12** FIFO page-replacement algorithm.

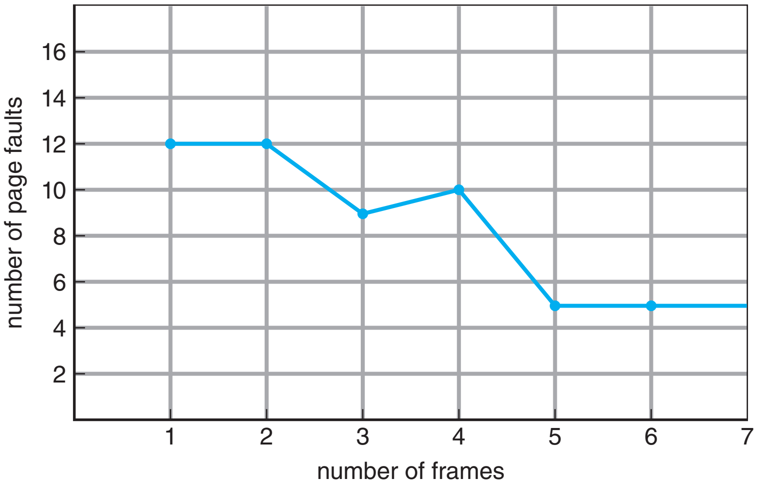


**Figure 10.13** Page-fault curve for FIFO replacement on a reference string.

one, a fault occurs almost immediately to retrieve the active page. Some other page must be replaced to bring the active page back into memory. Thus, a bad replacement choice increases the page-fault rate and slows process execution. It does not, however, cause incorrect execution.

To illustrate the problems that are possible with a FIFO page-replacement algorithm, consider the following reference string:

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Figure 10.13 shows the curve of page faults for this reference string versus the number of available frames. Notice that the number of faults for four frames (ten) is ***greater*** than the number of faults for three frames (nine)! This most unexpected result is known as **Belady's anomaly**: for some page-replacement algorithms, the page-fault rate may ***increase*** as the number of allocated frames increases. We would expect that giving more memory to a process would improve its performance. In some early research, investigators noticed that this assumption was not always true. Belady's anomaly was discovered as a result.

### 10.4.3 Optimal Page Replacement

One result of the discovery of Belady's anomaly was the search for an **optimal page-replacement algorithm**—the algorithm that has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly. Such an algorithm does exist and has been called OPT or MIN. It is simply this:

> Replace the page that will not be used for the longest period of time.

Use of this page-replacement algorithm guarantees the lowest possible page-fault rate for a fixed number of frames.

For example, on our sample reference string, the optimal page-replacement algorithm would yield nine page faults, as shown in Figure 10.14. The first three references cause faults that fill the three empty frames. The reference to page 2 replaces page 7, because page 7 will not be used until reference 18, whereas

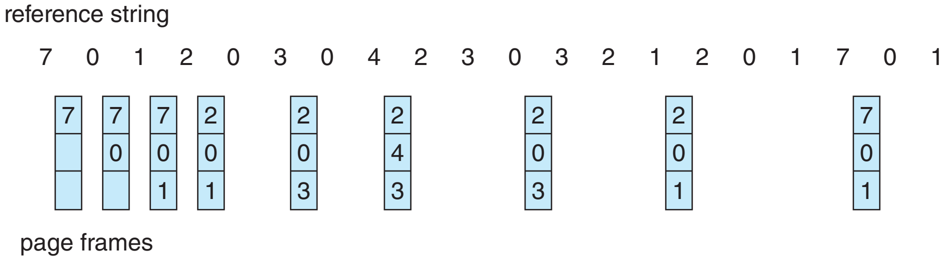


**Figure 10.14** Optimal page-replacement algorithm.

page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. With only nine page faults, optimal replacement is much better than a FIFO algorithm, which results in fifteen faults. (If we ignore the first three, which all algorithms must suffer, then optimal replacement is twice as good as FIFO replacement.) In fact, no replacement algorithm can process this reference string in three frames with fewer than nine faults.

Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string. (We encountered a similar situation with the SJF CPU-scheduling algorithm in Section 5.3.2.) As a result, the optimal algorithm is used mainly for comparison studies. For instance, it may be useful to know that, although a new algorithm is not optimal, it is within 12.3 percent of optimal at worst and within 4.7 percent on average.

### 10.4.4 LRU Page Replacement

If the optimal algorithm is not feasible, perhaps an approximation of the optimal algorithm is possible. The key distinction between the FIFO and OPT algorithms (other than looking backward versus forward in time) is that the FIFO algorithm uses the time when a page was brought into memory, whereas the OPT algorithm uses the time when a page is to be ***used.*** If we use the recent past as an approximation of the near future, then we can replace the page that ***has not been used*** for the longest period of time. This approach is the **least recently used (LRU) algorithm**.

LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time. We can think of this strategy as the optimal page-replacement algorithm looking backward in time, rather than forward. (Strangely, if we let $S^R$ be the reverse of a reference string $S$, then the page-fault rate for the OPT algorithm on $S$ is the same as the page-fault rate for the OPT algorithm on $S^R$. Similarly, the page-fault rate for the LRU algorithm on $S$ is the same as the page-fault rate for the LRU algorithm on $S^R$.)

The result of applying LRU replacement to our example reference string is shown in Figure 10.15. The LRU algorithm produces twelve faults. Notice that the first five faults are the same as those for optimal replacement. When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used. When it then faults

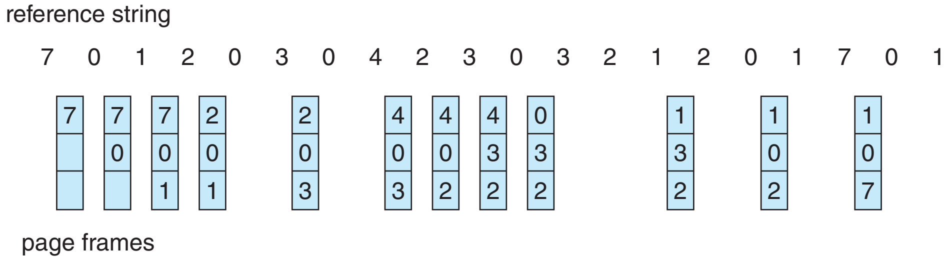


**Figure 10.15** LRU page-replacement algorithm.

for page 2, the LRU algorithm replaces page 3, since it is now the least recently used of the three pages in memory. Despite these problems, LRU replacement with twelve faults is much better than FIFO replacement with fifteen.

The LRU policy is often used as a page-replacement algorithm and is considered to be good. The major problem is ***how*** to implement LRU replacement. An LRU page-replacement algorithm may require substantial hardware assistance. The problem is to determine an order for the frames defined by the time of last use. Two implementations are feasible:

- **Counters**. In the simplest case, we associate with each page-table entry a time-of-use field and add to the CPU a logical clock or counter. The clock is incremented for every memory reference. Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page-table entry for that page. In this way, we always have the "time" of the last reference to each page. We replace the page with the smallest time value. This scheme requires a search of the page table to find the LRU page and a write to memory (to the time-of-use field in the page table) for each memory access. The times must also be maintained when page tables are changed (due to CPU scheduling). Overflow of the clock must be considered.
- **Stack**. Another approach to implementing LRU replacement is to keep a stack of page numbers. Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the most recently used page is always at the top of the stack, and the least recently used page is always at the bottom (Figure 10.16). Because entries must be removed from the middle of the stack, it is best to implement this approach by using a doubly linked list with a head pointer and a tail pointer. Removing a page and putting it on the top of the stack then requires changing six pointers at worst. Each update is a little more expensive, but there is no search for a replacement; the tail pointer points to the bottom of the stack, which is the LRU page. This approach is particularly appropriate for software or microcode implementations of LRU replacement.

Like optimal replacement, LRU replacement does not suffer from Belady's anomaly. Both belong to a class of page-replacement algorithms, called **stack algorithms**, that can never exhibit Belady's anomaly. A stack algorithm is an algorithm for which it can be shown that the set of pages in memory for $n$ frames is always a ***subset*** of the set of pages that would be in memory with $n$

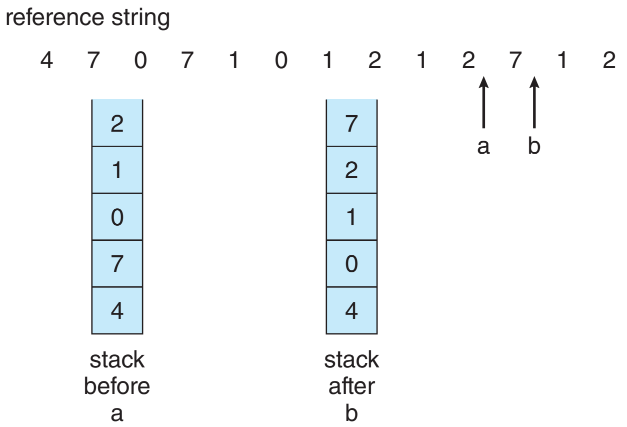


**Figure 10.16** Use of a stack to record the most recent page references.

+ 1 frames. For LRU replacement, the set of pages in memory would be the *n* most recently referenced pages. If the number of frames is increased, these *n* pages will still be the most recently referenced and so will still be in memory.

Note that neither implementation of LRU would be conceivable without hardware assistance beyond the standard TLB registers. The updating of the clock fields or stack must be done for ***every*** memory reference. If we were to use an interrupt for every reference to allow software to update such data structures, it would slow every memory reference by a factor of at least ten, hence slowing every process by a factor of ten. Few systems could tolerate that level of overhead for memory management.

### 10.4.5 LRU-Approximation Page Replacement

Not many computer systems provide sufficient hardware support for true LRU page replacement. In fact, some systems provide no hardware support, and other page-replacement algorithms (such as a FIFO algorithm) must be used. Many systems provide some help, however, in the form of a **reference bit**. The reference bit for a page is set by the hardware whenever that page is referenced (either a read or a write to any byte in the page). Reference bits are associated with each entry in the page table.

Initially, all bits are cleared (to 0) by the operating system. As a process executes, the bit associated with each page referenced is set (to 1) by the hardware. After some time, we can determine which pages have been used and which have not been used by examining the reference bits, although we do not know the ***order*** of use. This information is the basis for many page-replacement algorithms that approximate LRU replacement.

#### 10.4.5.1 Additional-Reference-Bits Algorithm

We can gain additional ordering information by recording the reference bits at regular intervals. We can keep an 8-bit byte for each page in a table in memory. At regular intervals (say, every 100 milliseconds), a timer interrupt transfers control to the operating system. The operating system shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right by 1 bit and discarding the low-order bit. These 8-bit shift registers contain the history of page use for the last eight time periods. If the shift register contains 00000000, for example, then the page has not been used for eight time periods. A page that is used at least once in each period has a shift register value of 11111111. A page with a history register value of 11000100 has been used more recently than one with a value of 01110111. If we interpret these 8-bit bytes as unsigned integers, the page with the lowest number is the LRU page, and it can be replaced. Notice that the numbers are not guaranteed to be unique, however. We can either replace (swap out) all pages with the smallest value or use the FIFO method to choose among them.

The number of bits of history included in the shift register can be varied, of course, and is selected (depending on the hardware available) to make the updating as fast as possible. In the extreme case, the number can be reduced to zero, leaving only the reference bit itself. This algorithm is called the **second-chance page-replacement algorithm**.

#### 10.4.5.2 Second-Chance Algorithm

The basic algorithm of second-chance replacement is a FIFO replacement algorithm. When a page has been selected, however, we inspect its reference bit. If the value is 0, we proceed to replace this page; but if the reference bit is set to 1, we give the page a second chance and move on to select the next FIFO page. When a page gets a second chance, its reference bit is cleared, and its arrival time is reset to the current time. Thus, a page that is given a second chance will not be replaced until all other pages have been replaced (or given second chances). In addition, if a page is used often enough to keep its reference bit set, it will never be replaced.

One way to implement the second-chance algorithm (sometimes referred to as the **clock** algorithm) is as a circular queue. A pointer (that is, a hand on the clock) indicates which page is to be replaced next. When a frame is needed, the pointer advances until it finds a page with a 0 reference bit. As it advances, it clears the reference bits (Figure 10.17). Once a victim page is found, the page is replaced, and the new page is inserted in the circular queue in that position. Notice that, in the worst case, when all bits are set, the pointer cycles through the whole queue, giving each page a second chance. It clears all the reference bits before selecting the next page for replacement. Second-chance replacement degenerates to FIFO replacement if all bits are set.

#### 10.4.5.3 Enhanced Second-Chance Algorithm

We can enhance the second-chance algorithm by considering the reference bit and the modify bit (described in Section 10.4.1) as an ordered pair. With these two bits, we have the following four possible classes:

1. (0, 0) neither recently used nor modified—best page to replace
2. (0, 1) not recently used but modified—not quite as good, because the page will need to be written out before replacement
3. (1, 0) recently used but clean—probably will be used again soon

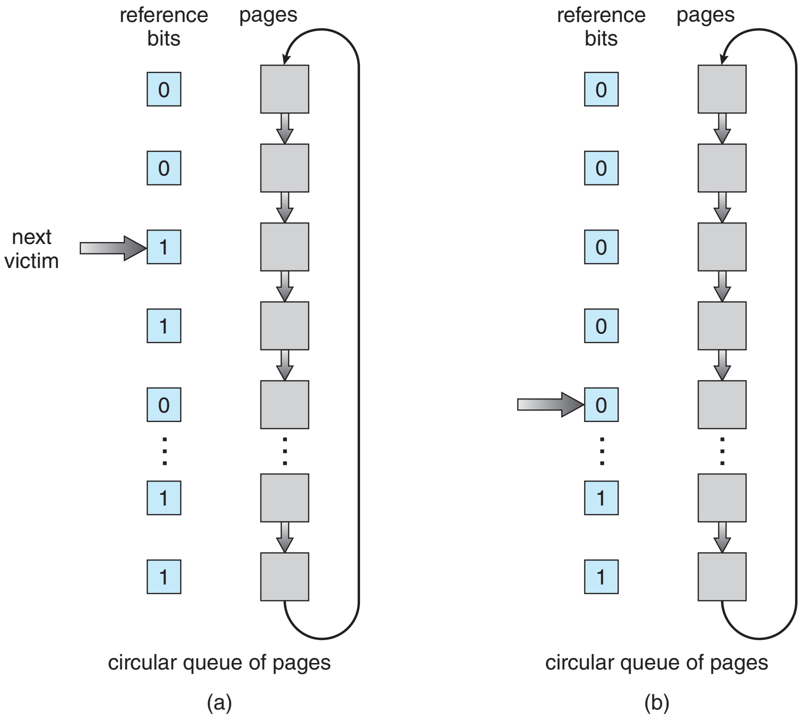


**Figure 10.17** Second-chance (clock) page-replacement algorithm.

4. (1, 1) recently used and modified—probably will be used again soon, and the page will be need to be written out to secondary storage before it can be replaced

Each page is in one of these four classes. When page replacement is called for, we use the same scheme as in the clock algorithm; but instead of examining whether the page to which we are pointing has the reference bit set to 1, we examine the class to which that page belongs. We replace the first page encountered in the lowest nonempty class. Notice that we may have to scan the circular queue several times before we find a page to be replaced. The major difference between this algorithm and the simpler clock algorithm is that here we give preference to those pages that have been modified in order to reduce the number of I/Os required.

### 10.4.6 Counting-Based Page Replacement

There are many other algorithms that can be used for page replacement. For example, we can keep a counter of the number of references that have been made to each page and develop the following two schemes.

- The **least frequently used** (LFU) page-replacement algorithm requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count. A problem arises, however, when a page is used heavily during the initial phase of a process but then is never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed. One solution is to shift the counts right by 1 bit at regular intervals, forming an exponentially decaying average usage count.

- The **most frequently used** (**MFU**) page-replacement algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

As you might expect, neither MFU nor LFU replacement is common. The implementation of these algorithms is expensive, and they do not approximate OPT replacement well.

### 10.4.7 Page-Buffering Algorithms

Other procedures are often used in addition to a specific page-replacement algorithm. For example, systems commonly keep a pool of free frames. When a page fault occurs, a victim frame is chosen as before. However, the desired page is read into a free frame from the pool before the victim is written out. This procedure allows the process to restart as soon as possible, without waiting for the victim page to be written out. When the victim is later written out, its frame is added to the free-frame pool.

An expansion of this idea is to maintain a list of modified pages. Whenever the paging device is idle, a modified page is selected and is written to secondary storage. Its modify bit is then reset. This scheme increases the probability that a page will be clean when it is selected for replacement and will not need to be written out.

Another modification is to keep a pool of free frames but to remember which page was in each frame. Since the frame contents are not modified when a frame is written to secondary storage, the old page can be reused directly from the free-frame pool if it is needed before that frame is reused. No I/O is needed in this case. When a page fault occurs, we first check whether the desired page is in the free-frame pool. If it is not, we must select a free frame and read into it.

Some versions of the UNIX system use this method in conjunction with the second-chance algorithm. It can be a useful augmentation to any page-replacement algorithm, to reduce the penalty incurred if the wrong victim page is selected. We describe these—and other—modifications in Section 10.5.3.

### 10.4.8 Applications and Page Replacement

In certain cases, applications accessing data through the operating system's virtual memory perform worse than if the operating system provided no buffering at all. A typical example is a database, which provides its own memory management and I/O buffering. Applications like this understand their memory use and storage use better than does an operating system that is implementing algorithms for general-purpose use. Furthermore, if the operating system is buffering I/O and the application is doing so as well, then twice the memory is being used for a set of I/O.

In another example, data warehouses frequently perform massive sequential storage reads, followed by computations and writes. The LRU algorithm would be removing old pages and preserving new ones, while the application would more likely be reading older pages than newer ones (as it starts its sequential reads again). Here, MFU would actually be more efficient than LRU.

Because of such problems, some operating systems give special programs the ability to use a secondary storage partition as a large sequential array of logical blocks, without any file-system data structures. This array is sometimes called the **raw disk**, and I/O to this array is termed raw I/O. Raw I/O bypasses all the file-system services, such as file I/O demand paging, file locking, prefetching, space allocation, file names, and directories. Note that although certain applications are more efficient when implementing their own special-purpose storage services on a raw partition, most applications perform better when they use the regular file-system services.

## 10.5 Allocation of Frames

We turn next to the issue of allocation. How do we allocate the fixed amount of free memory among the various processes? If we have 93 free frames and two processes, how many frames does each process get?

Consider a simple case of a system with 128 frames. The operating system may take 35, leaving 93 frames for the user process. Under pure demand paging, all 93 frames would initially be put on the free-frame list. When a user process started execution, it would generate a sequence of page faults. The first 93 page faults would all get free frames from the free-frame list. When the free-frame list was exhausted, a page-replacement algorithm would be used to select one of the 93 in-memory pages to be replaced with the 94th, and so on. When the process terminated, the 93 frames would once again be placed on the free-frame list.

There are many variations on this simple strategy. We can require that the operating system allocate all its buffer and table space from the free-frame list. When this space is not in use by the operating system, it can be used to support user paging. We can try to keep three free frames reserved on the free-frame list at all times. Thus, when a page fault occurs, there is a free frame available to page into. While the page swap is taking place, a replacement can be selected, which is then written to the storage device as the user process continues to execute. Other variants are also possible, but the basic strategy is clear: the user process is allocated any free frame.

### 10.5.1 Minimum Number of Frames

Our strategies for the allocation of frames are constrained in various ways. We cannot, for example, allocate more than the total number of available frames (unless there is page sharing). We must also allocate at least a minimum number of frames. Here, we look more closely at the latter requirement.

One reason for allocating at least a minimum number of frames involves performance. Obviously, as the number of frames allocated to each process decreases, the page-fault rate increases, slowing process execution. In addition, remember that, when a page fault occurs before an executing instruction is complete, the instruction must be restarted. Consequently, we must have enough frames to hold all the different pages that any single instruction can reference.

For example, consider a machine in which all memory-reference instructions may reference only one memory address. In this case, we need at least one frame for the instruction and one frame for the memory reference. In addition, if one-level indirect addressing is allowed (for example, a `load` instruction on frame 16 can refer to an address on frame 0, which is an indirect reference to frame 23), then paging requires at least three frames per process. (Think about what might happen if a process had only two frames.)

The minimum number of frames is defined by the computer architecture. For example, if the move instruction for a given architecture includes more than one word for some addressing modes, the instruction itself may straddle two frames. In addition, if each of its two operands may be indirect references, a total of six frames are required. As another example, the move instruction for Intel 32- and 64-bit architectures allows data to move only from register to register and between registers and memory; it does not allow direct memory-to-memory movement, thereby limiting the required minimum number of frames for a process.

Whereas the minimum number of frames per process is defined by the architecture, the maximum number is defined by the amount of available physical memory. In between, we are still left with significant choice in frame allocation.

### 10.5.2 Allocation Algorithms

The easiest way to split $m$ frames among $n$ processes is to give everyone an equal share, $m/n$ frames (ignoring frames needed by the operating system for the moment). For instance, if there are 93 frames and 5 processes, each process will get 18 frames. The 3 leftover frames can be used as a free-frame buffer pool. This scheme is called **equal allocation**.

An alternative is to recognize that various processes will need differing amounts of memory. Consider a system with a 1-KB frame size. If a small student process of 10 KB and an interactive database of 127 KB are the only two processes running in a system with 62 free frames, it does not make much sense to give each process 31 frames. The student process does not need more than 10 frames, so the other 21 are, strictly speaking, wasted.

To solve this problem, we can use **proportional allocation**, in which we allocate available memory to each process according to its size. Let the size of the virtual memory for process $p_i$ be $s_i$, and define

$$S = \sum s_i.$$

Then, if the total number of available frames is $m$, we allocate $a_i$ frames to process $p_i$, where $a_i$ is approximately

$$a_i = s_i/S \times m.$$

Of course, we must adjust each $a_i$ to be an integer that is greater than the minimum number of frames required by the instruction set, with a sum not exceeding $m$.

With proportional allocation, we would split 62 frames between two processes, one of 10 pages and one of 127 pages, by allocating 4 frames and 57 frames, respectively, since

$$10/137 \times 62 \approx 4 \text{ and}$$
$$127/137 \times 62 \approx 57.$$

In this way, both processes share the available frames according to their "needs," rather than equally.

In both equal and proportional allocation, of course, the allocation may vary according to the multiprogramming level. If the multiprogramming level is increased, each process will lose some frames to provide the memory needed for the new process. Conversely, if the multiprogramming level decreases, the frames that were allocated to the departed process can be spread over the remaining processes.

Notice that, with either equal or proportional allocation, a high-priority process is treated the same as a low-priority process. By its definition, however, we may want to give the high-priority process more memory to speed its execution, to the detriment of low-priority processes. One solution is to use a proportional allocation scheme wherein the ratio of frames depends not on the relative sizes of processes but rather on the priorities of processes or on a combination of size and priority.

### 10.5.3 Global versus Local Allocation

Another important factor in the way frames are allocated to the various processes is page replacement. With multiple processes competing for frames, we can classify page-replacement algorithms into two broad categories: **global replacement** and **local replacement**. Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process; that is, one process can take a frame from another. Local replacement requires that each process select from only its own set of allocated frames.

For example, consider an allocation scheme wherein we allow high-priority processes to select frames from low-priority processes for replacement. A process can select a replacement from among its own frames or the frames of any lower-priority process. This approach allows a high-priority process to increase its frame allocation at the expense of a low-priority process. Whereas with a local replacement strategy, the number of frames allocated to a process does not change, with global replacement, a process may happen to select only frames allocated to other processes, thus increasing the number of frames allocated to it (assuming that other processes do not choose ***its*** frames for replacement).

One problem with a global replacement algorithm is that the set of pages in memory for a process depends not only on the paging behavior of that process, but also on the paging behavior of other processes. Therefore, the same process may perform quite differently (for example, taking 0.5 seconds for one execution and 4.3 seconds for the next execution) because of totally external circumstances. Such is not the case with a local replacement algorithm. Under local replacement, the set of pages in memory for a process is affected by the paging behavior of only that process. Local replacement might hinder a process, however, by not making available to it other, less used pages of memory. Thus, global replacement generally results in greater system throughput. It is therefore the more commonly used method.

### *MAJOR AND MINOR PAGE FAULTS*

As described in Section 10.2.1, a page fault occurs when a page does not have a valid mapping in the address space of a process. Operating systems generally distinguish between two types of page faults: **major** and **minor** faults. (Windows refers to major and minor faults as **hard** and **soft** faults, respectively.) A major page fault occurs when a page is referenced and the page is not in memory. Servicing a major page fault requires reading the desired page from the backing store into a free frame and updating the page table. Demand paging typically generates an initially high rate of major page faults.

Minor page faults occur when a process does not have a logical mapping to a page, yet that page is in memory. Minor faults can occur for one of two reasons. First, a process may reference a shared library that is in memory, but the process does not have a mapping to it in its page table. In this instance, it is only necessary to update the page table to refer to the existing page in memory. A second cause of minor faults occurs when a page is reclaimed from a process and placed on the free-frame list, but the page has not yet been zeroed out and allocated to another process. When this kind of fault occurs, the frame is removed from the free-frame list and reassigned to the process. As might be expected, resolving a minor page fault is typically much less time consuming than resolving a major page fault.

You can observe the number of major and minor page faults in a Linux system using the command `ps -eo min_flt,maj_flt,cmd`, which outputs the number of minor and major page faults, as well as the command that launched the process. An example output of this ps command appears below:

```
MINFL      MAJFL      CMD
186509     32         /usr/lib/systemd/systemd-logind
76822      9          /usr/sbin/sshd -D
1937       0          vim 10.tex
699        14         /sbin/auditd -n
```

Here, it is interesting to note that, for most commands, the number of major page faults is generally quite low, whereas the number of minor faults is much higher. This indicates that Linux processes likely take significant advantage of shared libraries as, once a library is loaded in memory, subsequent page faults are only minor faults.

Next, we focus on one possible strategy that we can use to implement a global page-replacement policy. With this approach, we satisfy all memory requests from the free-frame list, but rather than waiting for the list to drop to zero before we begin selecting pages for replacement, we trigger page replacement when the list falls below a certain threshold. This strategy attempts to ensure there is always sufficient free memory to satisfy new requests.

Such a strategy is depicted in Figure 10.18. The strategy's purpose is to keep the amount of free memory above a minimum threshold. When it drops

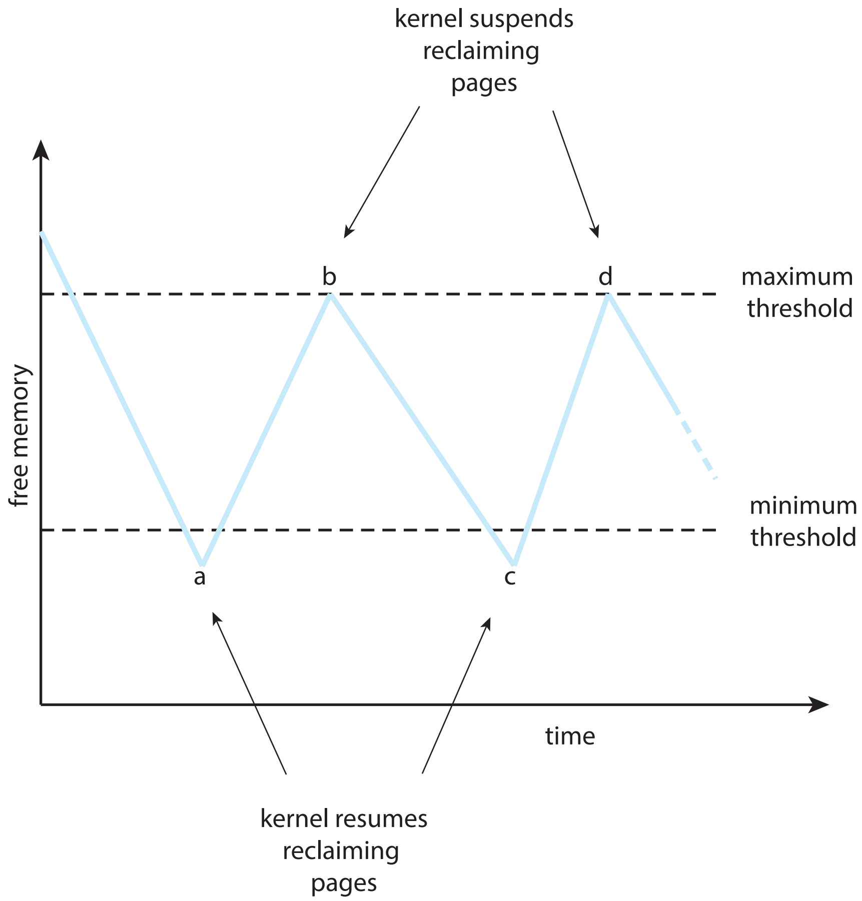


**Figure 10.18** Reclaiming pages.

below this threshold, a kernel routine is triggered that begins reclaiming pages from all processes in the system (typically excluding the kernel). Such kernel routines are often known as **reapers**, and they may apply any of the page-replacement algorithms covered in Section 10.4. When the amount of free memory reaches the maximum threshold, the reaper routine is suspended, only to resume once free memory again falls below the minimum threshold.

In Figure 10.18, we see that at point a the amount of free memory drops below the minimum threshold, and the kernel begins reclaiming pages and adding them to the free-frame list. It continues until the maximum threshold is reached (point b). Over time, there are additional requests for memory, and at point c the amount of free memory again falls below the minimum threshold. Page reclamation resumes, only to be suspended when the amount of free memory reaches the maximum threshold (point d). This process continues as long as the system is running.

As mentioned above, the kernel reaper routine may adopt any page-replacement algorithm, but typically it uses some form of LRU approximation. Consider what may happen, though, if the reaper routine is unable to maintain the list of free frames below the minimum threshold. Under these circumstances, the reaper routine may begin to reclaim pages more aggressively. For example, perhaps it will suspend the second-chance algorithm and use pure FIFO. Another, more extreme, example occurs in Linux; when the amount of free memory falls to ***very*** low levels, a routine known as the **out-of-memory** (**OOM**) **killer** selects a process to terminate, thereby freeing its memory. How does Linux determine which process to terminate? Each process has what is known as an OOM score, with a higher score increasing the likelihood that the process could be terminated by the OOM killer routine. OOM scores are calculated according to the percentage of memory a process is using—the higher the percentage, the higher the OOM score. (OOM scores can be viewed in the `/proc` file system, where the score for a process with `pid` 2500 can be viewed as `/proc/2500/oom_score`.)

In general, not only can reaper routines vary how aggressively they reclaim memory, but the values of the minimum and maximum thresholds can be varied as well. These values can be set to default values, but some systems may allow a system administrator to configure them based on the amount of physical memory in the system.

### 10.5.4 Non-Uniform Memory Access

Thus far in our coverage of virtual memory, we have assumed that all main memory is created equal—or at least that it is accessed equally. On **non-uniform memory access** (**NUMA**) systems with multiple CPUs (Section 1.3.2), that is not the case. On these systems, a given CPU can access some sections of main memory faster than it can access others. These performance differences are caused by how CPUs and memory are interconnected in the system. Such a system is made up of multiple CPUs, each with its own local memory (Figure 10.19). The CPUs are organized using a shared system interconnect, and as you might expect, a CPU can access its local memory faster than memory local to another CPU. NUMA systems are without exception slower than systems in which all accesses to main memory are treated equally. However, as described in Section 1.3.2, NUMA systems can accommodate more CPUs and therefore achieve greater levels of throughput and parallelism.

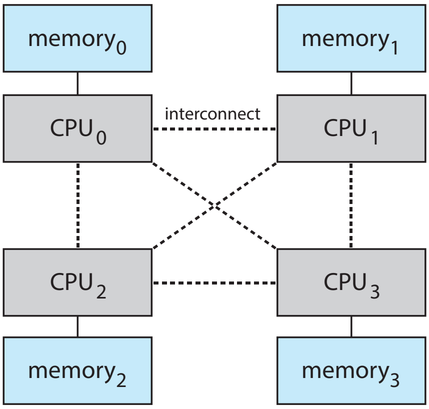


**Figure 10.19** NUMA multiprocessing architecture.

Managing which page frames are stored at which locations can significantly affect performance in NUMA systems. If we treat memory as uniform in such a system, CPUs may wait significantly longer for memory access than if we modify memory allocation algorithms to take NUMA into account. We described some of these modifications in Section 5.5.4. Their goal is to have memory frames allocated "as close as possible" to the CPU on which the process is running. (The definition of ***close*** is "with minimum latency," which typically means on the same system board as the CPU). Thus, when a process incurs a page fault, a NUMA-aware virtual memory system will allocate that process a frame as close as possible to the CPU on which the process is running.

To take NUMA into account, the scheduler must track the last CPU on which each process ran. If the scheduler tries to schedule each process onto its previous CPU, and the virtual memory system tries to allocate frames for the process close to the CPU on which it is being scheduled, then improved cache hits and decreased memory access times will result.

The picture is more complicated once threads are added. For example, a process with many running threads may end up with those threads scheduled on many different system boards. How should the memory be allocated in this case?

As we discussed in Section 5.7.1, Linux manages this situation by having the kernel identify a hierarchy of scheduling domains. The Linux CFS scheduler does not allow threads to migrate across different domains and thus incur memory access penalties. Linux also has a separate free-frame list for each NUMA node, thereby ensuring that a thread will be allocated memory from the node on which it is running. Solaris solves the problem similarly by creating **lgroups** (for "locality groups") in the kernel. Each lgroup gathers together CPUs and memory, and each CPU in that group can access any memory in the group within a defined latency interval. In addition, there is a hierarchy of lgroups based on the amount of latency between the groups, similar to the hierarchy of scheduling domains in Linux. Solaris tries to schedule all threads of a process and allocate all memory of a process within an lgroup. If that is not possible, it picks nearby lgroups for the rest of the resources needed. This practice minimizes overall memory latency and maximizes CPU cache hit rates.

## 10.6 Thrashing

Consider what occurs if a process does not have "enough" frames—that is, it does not have the minimum number of frames it needs to support pages in the working set. The process will quickly page-fault. At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately.

This high paging activity is called **thrashing**. A process is thrashing if it is spending more time paging than executing. As you might expect, thrashing results in severe performance problems.

### 10.6.1 Cause of Thrashing

Consider the following scenario, which is based on the actual behavior of early paging systems. The operating system monitors CPU utilization. If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new process to the system. A global page-replacement algorithm is used; it replaces pages without regard to the process to which they belong. Now suppose that a process enters a new phase in its execution and needs more frames. It starts faulting and taking frames away from other processes. These processes need those pages, however, and so they also fault, taking frames from other processes. These faulting processes must use the paging device to swap pages in and out. As they queue up for the paging device, the ready queue empties. As processes wait for the paging device, CPU utilization decreases.

The CPU scheduler sees the decreasing CPU utilization and ***increases*** the degree of multiprogramming as a result. The new process tries to get started by taking frames from running processes, causing more page faults and a longer queue for the paging device. As a result, CPU utilization drops even further, and the CPU scheduler tries to increase the degree of multiprogramming even more. Thrashing has occurred, and system throughput plunges. The page-fault rate increases tremendously. As a result, the effective memory-access time increases. No work is getting done, because the processes are spending all their time paging.

This phenomenon is illustrated in Figure 10.20, in which CPU utilization is plotted against the degree of multiprogramming. As the degree of multiprogramming increases, CPU utilization also increases, although more slowly, until a maximum is reached. If the degree of multiprogramming is increased further, thrashing sets in, and CPU utilization drops sharply. At this point, to increase CPU utilization and stop thrashing, we must ***decrease*** the degree of multiprogramming.

We can limit the effects of thrashing by using a **local replacement algorithm** (or **priority replacement algorithm**). As mentioned earlier, local replacement requires that each process select from only its own set of allocated frames. Thus, if one process starts thrashing, it cannot steal frames from another process and cause the latter to thrash as well. However, the problem is not entirely solved. If processes are thrashing, they will be in the queue for the paging device most of the time. The average service time for a page fault will increase

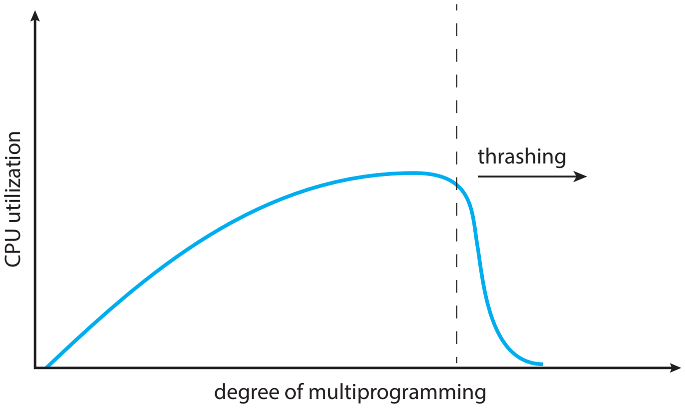


**Figure 10.20** Thrashing.

because of the longer average queue for the paging device. Thus, the effective access time will increase even for a process that is not thrashing.

To prevent thrashing, we must provide a process with as many frames as it needs. But how do we know how many frames it “needs”? One strategy starts by looking at how many frames a process is actually using. This approach defines the **locality model** of process execution.

The locality model states that, as a process executes, it moves from locality to locality. A locality is a set of pages that are actively used together. A running program is generally composed of several different localities, which may overlap. For example, when a function is called, it defines a new locality. In this

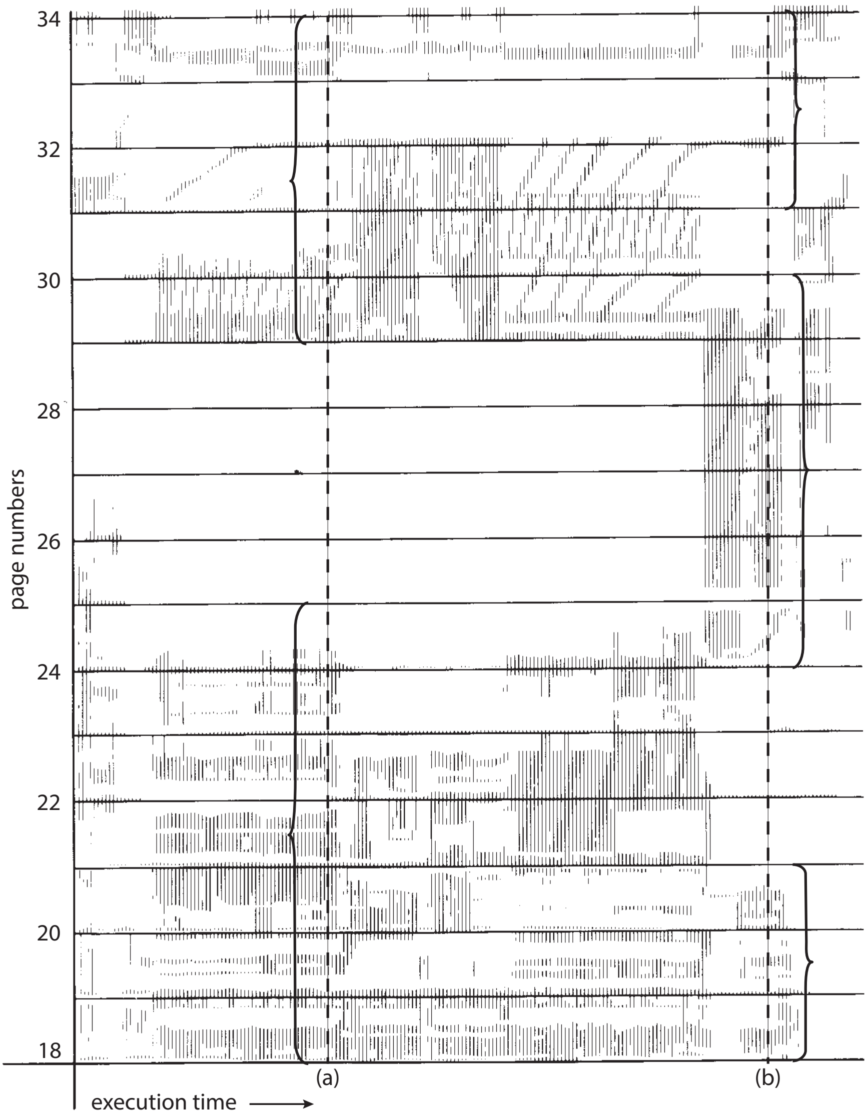


**Figure 10.21** Locality in a memory-reference pattern.

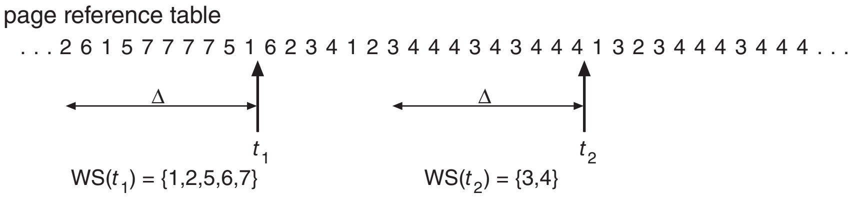


**Figure 10.22** Working-set model.

locality, memory references are made to the instructions of the function call, its local variables, and a subset of the global variables. When we exit the function, the process leaves this locality, since the local variables and instructions of the function are no longer in active use. We may return to this locality later.

Figure 10.21 illustrates the concept of locality and how a process's locality changes over time. At time (*a*), the locality is the set of pages $\{18, 19, 20, 21, 22, 23, 24, 29, 30, 33\}$. At time (*b*), the locality changes to $\{18, 19, 20, 24, 25, 26, 27, 28, 29, 31, 32, 33\}$. Notice the overlap, as some pages (for example, 18, 19, and 20) are part of both localities.

Thus, we see that localities are defined by the program structure and its data structures. The locality model states that all programs will exhibit this basic memory reference structure. Note that the locality model is the unstated principle behind the caching discussions so far in this book. If accesses to any types of data were random rather than patterned, caching would be useless.

Suppose we allocate enough frames to a process to accommodate its current locality. It will fault for the pages in its locality until all these pages are in memory; then, it will not fault again until it changes localities. If we do not allocate enough frames to accommodate the size of the current locality, the process will thrash, since it cannot keep in memory all the pages that it is actively using.

### 10.6.2 Working-Set Model

The **working-set model** is based on the assumption of locality. This model uses a parameter, $\Delta$, to define the **working-set window**. The idea is to examine the most recent $\Delta$ page references. The set of pages in the most recent $\Delta$ page references is the **working set** (Figure 10.22). If a page is in active use, it will be in the working set. If it is no longer being used, it will drop from the working set $\Delta$ time units after its last reference. Thus, the working set is an approximation of the program's locality.

For example, given the sequence of memory references shown in Figure 10.22, if $\Delta = 10$ memory references, then the working set at time $t_1$ is $\{1, 2, 5, 6, 7\}$. By time $t_2$, the working set has changed to $\{3, 4\}$.

The accuracy of the working set depends on the selection of $\Delta$. If $\Delta$ is too small, it will not encompass the entire locality; if $\Delta$ is too large, it may overlap several localities. In the extreme, if $\Delta$ is infinite, the working set is the set of pages touched during the process execution.

The most important property of the working set, then, is its size. If we compute the working-set size, $WSS_i$, for each process in the system, we can then consider that

$$D = \sum WSS_i,$$

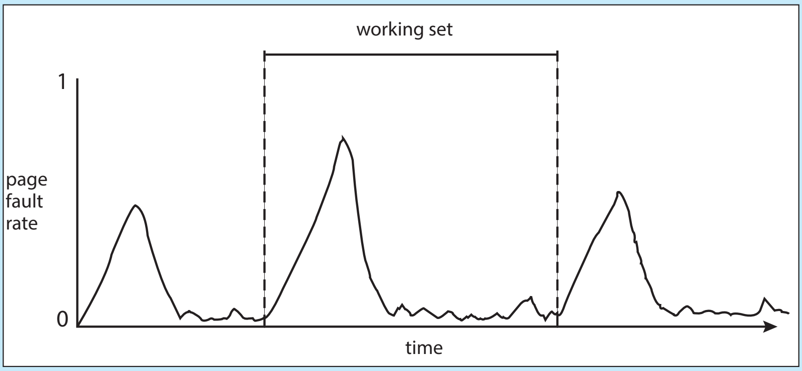
where $D$ is the total demand for frames. Each process is actively using the pages in its working set. Thus, process $i$ needs $WSS_i$ frames. If the total demand is greater than the total number of available frames ($D > m$), thrashing will occur, because some processes will not have enough frames.

Once $\Delta$ has been selected, use of the working-set model is simple. The operating system monitors the working set of each process and allocates to that working set enough frames to provide it with its working-set size. If there are enough extra frames, another process can be initiated. If the sum of the working-set sizes increases, exceeding the total number of available frames, the operating system selects a process to suspend. The process's pages are written out (swapped), and its frames are reallocated to other processes. The suspended process can be restarted later.

This working-set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible. Thus, it optimizes CPU utilization. The difficulty with the working-set model is keeping track of the working set. The

### *WORKING SETS AND PAGE-FAULT RATES*

There is a direct relationship between the working set of a process and its page-fault rate. Typically, as shown in Figure 10.22, the working set of a process changes over time as references to data and code sections move from one locality to another. Assuming there is sufficient memory to store the working set of a process (that is, the process is not thrashing), the page-fault rate of the process will transition between peaks and valleys over time. This general behavior is shown below:



A peak in the page-fault rate occurs when we begin demand-paging a new locality. However, once the working set of this new locality is in memory, the page-fault rate falls. When the process moves to a new working set, the page-fault rate rises toward a peak once again, returning to a lower rate once the new working set is loaded into memory. The span of time between the start of one peak and the start of the next peak represents the transition from one working set to another.

working-set window is a moving window. At each memory reference, a new reference appears at one end, and the oldest reference drops off the other end. A page is in the working set if it is referenced anywhere in the working-set window.

We can approximate the working-set model with a fixed-interval timer interrupt and a reference bit. For example, assume that $\Delta$ equals 10,000 references and that we can cause a timer interrupt every 5,000 references. When we get a timer interrupt, we copy and clear the reference-bit values for each page. Thus, if a page fault occurs, we can examine the current reference bit and two in-memory bits to determine whether a page was used within the last 10,000 to 15,000 references. If it was used, at least one of these bits will be on. If it has not been used, these bits will be off. Pages with at least one bit on will be considered to be in the working set.

Note that this arrangement is not entirely accurate, because we cannot tell where, within an interval of 5,000, a reference occurred. We can reduce the uncertainty by increasing the number of history bits and the frequency of interrupts (for example, 10 bits and interrupts every 1,000 references). However, the cost to service these more frequent interrupts will be correspondingly higher.

### 10.6.3 Page-Fault Frequency

The working-set model is successful, and knowledge of the working set can be useful for prepaging (Section 10.9.1), but it seems a clumsy way to control thrashing. A strategy that uses the **page-fault frequency** (**PFF**) takes a more direct approach.

The specific problem is how to prevent thrashing. Thrashing has a high page-fault rate. Thus, we want to control the page-fault rate. When it is too high, we know that the process needs more frames. Conversely, if the page-fault rate is too low, then the process may have too many frames. We can establish upper and lower bounds on the desired page-fault rate (Figure 10.23). If the actual page-fault rate exceeds the upper limit, we allocate the process

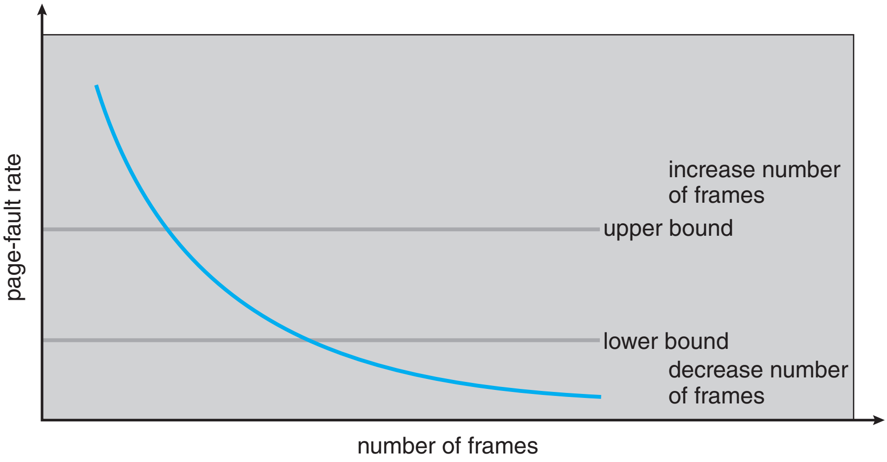


**Figure 10.23** Page-fault frequency.

another frame. If the page-fault rate falls below the lower limit, we remove a frame from the process. Thus, we can directly measure and control the page-fault rate to prevent thrashing.

As with the working-set strategy, we may have to swap out a process. If the page-fault rate increases and no free frames are available, we must select some process and swap it out to backing store. The freed frames are then distributed to processes with high page-fault rates.

### 10.6.4 Current Practice

Practically speaking, thrashing and the resulting swapping have a disagreeably high impact on performance. The current best practice in implementing a computer system is to include enough physical memory, whenever possible, to avoid thrashing and swapping. From smartphones through large servers, providing enough memory to keep all working sets in memory concurrently, except under extreme conditions, provides the best user experience.

## 10.7 Memory Compression

An alternative to paging is **memory compression**. Here, rather than paging out modified frames to swap space, we compress several frames into a single frame, enabling the system to reduce memory usage without resorting to swapping pages.

In Figure 10.24, the free-frame list contains six frames. Assume that this number of free frames falls below a certain threshold that triggers page replacement. The replacement algorithm (say, an LRUapproximation algorithm) selects four frames—15, 3, 35, and 26—to place on the free-frame list. It first places these frames on a modified-frame list. Typically, the modified-frame list would next be written to swap space, making the frames available to the free-frame list. An alternative strategy is to compress a number of frames—say, three—and store their compressed versions in a single page frame.

In Figure 10.25, frame 7 is removed from the free-frame list. Frames 15, 3, and 35 are compressed and stored in frame 7, which is then stored in the list of compressed frames. The frames 15, 3, and 35 can now be moved to the free-frame list. If one of the three compressed frames is later referenced, a page fault occurs, and the compressed frame is decompressed, restoring the three pages 15, 3, and 35 in memory.

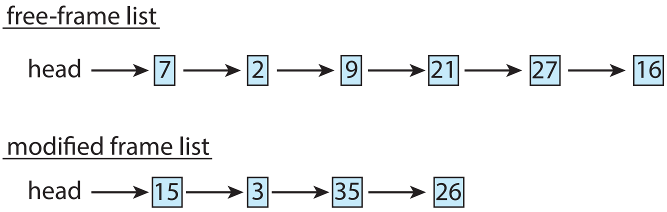


**Figure 10.24** Free-frame list before compression.

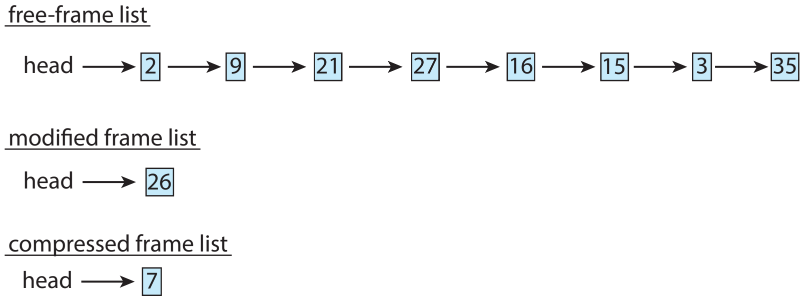


**Figure 10.25** Free-frame list after compression

As we have noted, mobile systems generally do not support either standard swapping or swapping pages. Thus, memory compression is an integral part of the memory-management strategy for most mobile operating systems, including Android and iOS. In addition, both Windows 10 and macOS support memory compression. For Windows 10, Microsoft developed the **Universal Windows Platform** (**UWP**) architecture, which provides a common app platform for devices that run Windows 10, including mobile devices. UWP apps running on mobile devices are candidates for memory compression. macOS first supported memory compression with Version 10.9 of the operating system, first compressing LRU pages when free memory is short and then paging if that doesn't solve the problem. Performance tests indicate that memory compression is faster than paging even to SSD secondary storage on laptop and desktop macOS systems.

Although memory compression does require allocating free frames to hold the compressed pages, a significant memory saving can be realized, depending on the reductions achieved by the compression algorithm. (In the example above, the three frames were reduced to one-third of their original size.) As with any form of data compression, there is contention between the speed of the compression algorithm and the amount of reduction that can be achieved (known as the **compression ratio**). In general, higher compression ratios (greater reductions) can be achieved by slower, more computationally expensive algorithms. Most algorithms in use today balance these two factors, achieving relatively high compression ratios using fast algorithms. In addition, compression algorithms have improved by taking advantage of multiple computing cores and performing compression in parallel. For example, Microsoft's Xpress and Apple's WKdm compression algorithms are considered fast, and they report compressing pages to 30 to 50 percent of their original size.

## 10.8 Allocating Kernel Memory

When a process running in user mode requests additional memory, pages are allocated from the list of free page frames maintained by the kernel. This list is typically populated using a page-replacement algorithm such as those discussed in Section 10.4 and most likely contains free pages scattered throughout physical memory, as explained earlier. Remember, too, that if a user process requests a single byte of memory, internal fragmentation will result, as the process will be granted an entire page frame.

Kernel memory is often allocated from a free-memory pool different from the list used to satisfy ordinary user-mode processes. There are two primary reasons for this:

1. The kernel requests memory for data structures of varying sizes, some of which are less than a page in size. As a result, the kernel must use memory conservatively and attempt to minimize waste due to fragmentation. This is especially important because many operating systems do not subject kernel code or data to the paging system.
2. Pages allocated to user-mode processes do not necessarily have to be in contiguous physical memory. However, certain hardware devices interact directly with physical memory—without the benefit of a virtual memory interface—and consequently may require memory residing in physically contiguous pages.

In the following sections, we examine two strategies for managing free memory that is assigned to kernel processes: the "buddy system" and slab allocation.

### 10.8.1 Buddy System

The buddy system allocates memory from a fixed-size segment consisting of physically contiguous pages. Memory is allocated from this segment using a **power-of-2 allocator**, which satisfies requests in units sized as a power of 2 (4 KB, 8 KB, 16 KB, and so forth). A request in units not appropriately sized is rounded up to the next highest power of 2. For example, a request for 11 KB is satisfied with a 16-KB segment.

Let's consider a simple example. Assume the size of a memory segment is initially 256 KB and the kernel requests 21 KB of memory. The segment is initially divided into two **buddies**—which we will call $A_L$ and $A_R$—each 128 KB in size. One of these buddies is further divided into two 64-KB buddies—$B_L$ and $B_R$. However, the next-highest power of 2 from 21 KB is 32 KB so either $B_L$ or $B_R$ is again divided into two 32-KB buddies, $C_L$ and $C_R$. One of these buddies is used to satisfy the 21-KB request. This scheme is illustrated in Figure 10.26, where $C_L$ is the segment allocated to the 21-KB request.

An advantage of the buddy system is how quickly adjacent buddies can be combined to form larger segments using a technique known as **coalescing**. In Figure 10.26, for example, when the kernel releases the $C_L$ unit it was allocated, the system can coalesce $C_L$ and $C_R$ into a 64-KB segment. This segment, $B_L$, can in turn be coalesced with its buddy $B_R$ to form a 128-KB segment. Ultimately, we can end up with the original 256-KB segment.

The obvious drawback to the buddy system is that rounding up to the next highest power of 2 is very likely to cause fragmentation within allocated segments. For example, a 33-KB request can only be satisfied with a 64-KB segment. In fact, we cannot guarantee that less than 50 percent of the allocated unit will be wasted due to internal fragmentation. In the following section, we explore a memory allocation scheme where no space is lost due to fragmentation.

### 10.8.2 Slab Allocation

A second strategy for allocating kernel memory is known as **slab allocation**. A **slab** is made up of one or more physically contiguous pages. A **cache** consists

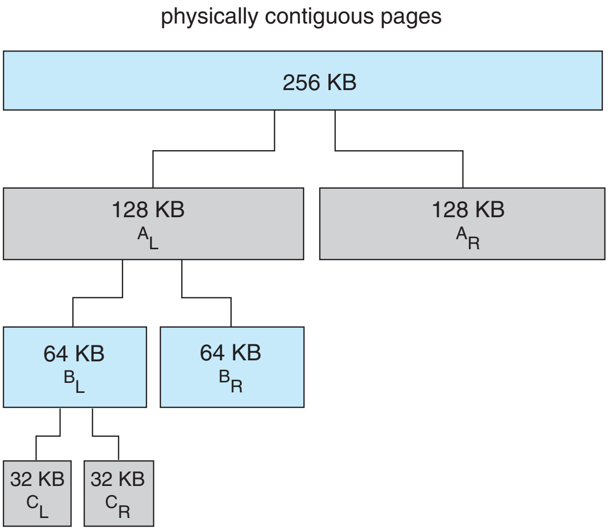

physically contiguous pages
256 KB
128 KB
$A_L$
128 KB
$A_R$
64 KB
$B_L$
64 KB
$B_R$
32 KB
$C_L$
32 KB
$C_R$

**Figure 10.26** Buddy system allocation.

of one or more slabs. There is a single cache for each unique kernel data structure—for example, a separate cache for the data structure representing process descriptors, a separate cache for file objects, a separate cache for semaphores, and so forth. Each cache is populated with **objects** that are instantiations of the kernel data structure the cache represents. For example, the cache representing semaphores stores instances of semaphore objects, the cache representing process descriptors stores instances of process descriptor objects, and so forth. The relationship among slabs, caches, and objects is shown in Figure 10.27. The figure shows two kernel objects 3 KB in size and three objects 7 KB in size, each stored in a separate cache.

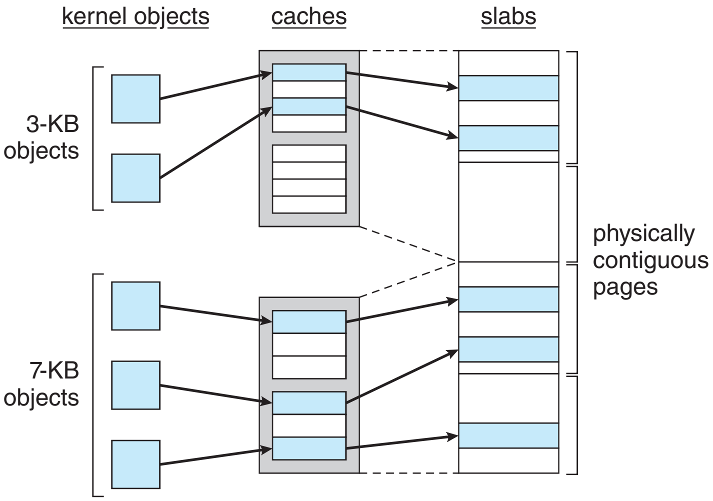

kernel objects
caches
slabs
3-KB objects
7-KB objects
physically contiguous pages

**Figure 10.27** Slab allocation.

The slab-allocation algorithm uses caches to store kernel objects. When a cache is created, a number of objects—which are initially marked as `free`—are allocated to the cache. The number of objects in the cache depends on the size of the associated slab. For example, a 12-KB slab (made up of three contiguous 4-KB pages) could store six 2-KB objects. Initially, all objects in the cache are marked as free. When a new object for a kernel data structure is needed, the allocator can assign any free object from the cache to satisfy the request. The object assigned from the cache is marked as `used`.

Let's consider a scenario in which the kernel requests memory from the slab allocator for an object representing a process descriptor. In Linux systems, a process descriptor is of the type `struct task_struct`, which requires approximately 1.7 KB of memory. When the Linux kernel creates a new task, it requests the necessary memory for the `struct task_struct` object from its cache. The cache will fulfill the request using a `struct task_struct` object that has already been allocated in a slab and is marked as free.

In Linux, a slab may be in one of three possible states:

1. **Full**. All objects in the slab are marked as used.
2. **Empty**. All objects in the slab are marked as free.
3. **Partial**. The slab consists of both used and free objects.

The slab allocator first attempts to satisfy the request with a free object in a partial slab. If none exists, a free object is assigned from an empty slab. If no empty slabs are available, a new slab is allocated from contiguous physical pages and assigned to a cache; memory for the object is allocated from this slab.

The slab allocator provides two main benefits:

1. No memory is wasted due to fragmentation. Fragmentation is not an issue because each unique kernel data structure has an associated cache, and each cache is made up of one or more slabs that are divided into chunks the size of the objects being represented. Thus, when the kernel requests memory for an object, the slab allocator returns the exact amount of memory required to represent the object.
2. Memory requests can be satisfied quickly. The slab allocation scheme is thus particularly effective for managing memory when objects are frequently allocated and deallocated, as is often the case with requests from the kernel. The act of allocating—and releasing—memory can be a time-consuming process. However, objects are created in advance and thus can be quickly allocated from the cache. Furthermore, when the kernel has finished with an object and releases it, it is marked as free and returned to its cache, thus making it immediately available for subsequent requests from the kernel.

The slab allocator first appeared in the Solaris 2.4 kernel. Because of its general-purpose nature, this allocator is now also used for certain user-mode memory requests in Solaris. Linux originally used the buddy system; however, beginning with Version 2.2, the Linux kernel adopted the slab allocator. Linux refers to its slab implementation as SLAB. Recent distributions of Linux include two other kernel memory allocators—the SLOB and SLUB allocators.

The SLOB allocator is designed for systems with a limited amount of memory, such as embedded systems. SLOB (which stands for "simple list of blocks") maintains three lists of objects: ***small*** (for objects less than 256 bytes), ***medium*** (for objects less than 1,024 bytes), and ***large*** (for all other objects less than the size of a page). Memory requests are allocated from an object on the appropriate list using a first-fit policy.

Beginning with Version 2.6.24, the SLUB allocator replaced SLAB as the default allocator for the Linux kernel. SLUB reduced much of the overhead required by the SLAB allocator. For instance, whereas SLAB stores certain metadata with each slab, SLUB stores these data in the `page` structure the Linux kernel uses for each page. Additionally, SLUB does not include the per-CPU queues that the SLAB allocator maintains for objects in each cache. For systems with a large number of processors, the amount of memory allocated to these queues is significant. Thus, SLUB provides better performance as the number of processors on a system increases.

## 10.9 Other Considerations

The major decisions that we make for a paging system are the selections of a replacement algorithm and an allocation policy, which we discussed earlier in this chapter. There are many other considerations as well, and we discuss several of them here.

### 10.9.1 Prepaging

An obvious property of pure demand paging is the large number of page faults that occur when a process is started. This situation results from trying to get the initial locality into memory. **Prepaging** is an attempt to prevent this high level of initial paging. The strategy is to bring some—or all—of the pages that will be needed into memory at one time.

In a system using the working-set model, for example, we could keep with each process a list of the pages in its working set. If we must suspend a process (due to a lack of free frames), we remember the working set for that process. When the process is to be resumed (because I/O has finished or enough free frames have become available), we automatically bring back into memory its entire working set before restarting the process.

Prepaging may offer an advantage in some cases. The question is simply whether the cost of using prepaging is less than the cost of servicing the corresponding page faults. It may well be the case that many of the pages brought back into memory by prepaging will not be used.

Assume that $s$ pages are prepaged and a fraction $\alpha$ of these $s$ pages is actually used ($0 \leq \alpha \leq 1$). The question is whether the cost of the $s * \alpha$ saved page faults is greater or less than the cost of prepaging $s * (1 - \alpha)$ unnecessary pages. If $\alpha$ is close to 0, prepaging loses; if $\alpha$ is close to 1, prepaging wins.

Note also that prepaging an executable program may be difficult, as it may be unclear exactly what pages should be brought in. Prepaging a file may be more predictable, since files are often accessed sequentially. The Linux `readahead()` system call prefetches the contents of a file into memory so that subsequent accesses to the file will take place in main memory.

### 10.9.2 Page Size

The designers of an operating system for an existing machine seldom have a choice concerning the page size. However, when new machines are being designed, a decision regarding the best page size must be made. As you might expect, there is no single best page size. Rather, there is a set of factors that support various sizes. Page sizes are invariably powers of 2, generally ranging from 4,096 ($2^{12}$) to 4,194,304 ($2^{22}$) bytes.

How do we select a page size? One concern is the size of the page table. For a given virtual memory space, decreasing the page size increases the number of pages and hence the size of the page table. For a virtual memory of 4 MB ($2^{22}$), for example, there would be 4,096 pages of 1,024 bytes but only 512 pages of 8,192 bytes. Because each active process must have its own copy of the page table, a large page size is desirable.

Memory is better utilized with smaller pages, however. If a process is allocated memory starting at location 00000 and continuing until it has as much as it needs, it probably will not end exactly on a page boundary. Thus, a part of the final page must be allocated (because pages are the units of allocation) but will be unused (creating internal fragmentation). Assuming independence of process size and page size, we can expect that, on the average, half of the final page of each process will be wasted. This loss is only 256 bytes for a page of 512 bytes but is 4,096 bytes for a page of 8,192 bytes. To minimize internal fragmentation, then, we need a small page size.

Another problem is the time required to read or write a page. As you will see in Section 11.1, when the storage device is an HDD, I/O time is composed of seek, latency, and transfer times. Transfer time is proportional to the amount transferred (that is, the page size)—a fact that would seem to argue for a small page size. However, latency and seek time normally dwarf transfer time. At a transfer rate of 50 MB per second, it takes only 0.01 milliseconds to transfer 512 bytes. Latency time, though, is perhaps 3 milliseconds, and seek time 5 milliseconds. Of the total I/O time (8.01 milliseconds), therefore, only about 0.1 percent is attributable to the actual transfer. Doubling the page size increases I/O time to only 8.02 milliseconds. It takes 8.02 milliseconds to read a single page of 1,024 bytes but 16.02 milliseconds to read the same amount as two pages of 512 bytes each. Thus, a desire to minimize I/O time argues for a larger page size.

With a smaller page size, though, total I/O should be reduced, since locality will be improved. A smaller page size allows each page to match program locality more accurately. For example, consider a process 200 KB in size, of which only half (100 KB) is actually used in an execution. If we have only one large page, we must bring in the entire page, a total of 200 KB transferred and allocated. If instead we had pages of only 1 byte, then we could bring in only the 100 KB that are actually used, resulting in only 100 KB transferred and allocated. With a smaller page size, then, we have better **resolution**, allowing us to isolate only the memory that is actually needed. With a larger page size, we must allocate and transfer not only what is needed but also anything else that happens to be in the page, whether it is needed or not. Thus, a smaller page size should result in less I/O and less total allocated memory.

But did you notice that with a page size of 1 byte, we would have a page fault for ***each*** byte? A process of 200 KB that used only half of that memory would generate only one page fault with a page size of 200 KB but 102,400 page faults with a page size of 1 byte. Each page fault generates the large amount of overhead needed for processing the interrupt, saving registers, replacing a page, queuing for the paging device, and updating tables. To minimize the number of page faults, we need to have a large page size.

Other factors must be considered as well (such as the relationship between page size and sector size on the paging device). The problem has no best answer. As we have seen, some factors (internal fragmentation, locality) argue for a small page size, whereas others (table size, I/O time) argue for a large page size. Nevertheless, the historical trend is toward larger page sizes, even for mobile systems. Indeed, the first edition of *Operating System Concepts* (1983) used 4,096 bytes as the upper bound on page sizes, and this value was the most common page size in 1990. Modern systems may now use much larger page sizes, as you will see in the following section.

### 10.9.3 TLB Reach

In Chapter 9, we introduced the **hit ratio** of the TLB. Recall that the hit ratio for the TLB refers to the percentage of virtual address translations that are resolved in the TLB rather than the page table. Clearly, the hit ratio is related to the number of entries in the TLB, and the way to increase the hit ratio is by increasing the number of entries. This, however, does not come cheaply, as the associative memory used to construct the TLB is both expensive and power hungry.

Related to the hit ratio is a similar metric: the **TLB reach**. The TLB reach refers to the amount of memory accessible from the TLB and is simply the number of entries multiplied by the page size. Ideally, the working set for a process is stored in the TLB. If it is not, the process will spend a considerable amount of time resolving memory references in the page table rather than the TLB. If we double the number of entries in the TLB, we double the TLB reach. However, for some memory-intensive applications, this may still prove insufficient for storing the working set.

Another approach for increasing the TLB reach is to either increase the size of the page or provide multiple page sizes. If we increase the page size—say, from 4 KB to 16 KB—we quadruple the TLB reach. However, this may lead to an increase in fragmentation for some applications that do not require such a large page size. Alternatively, most architectures provide support for more than one page size, and an operating system can be configured to take advantage of this support. For example, the default page size on Linux systems is 4 KB; however, Linux also provides **huge pages**, a feature that designates a region of physical memory where larger pages (for example, 2 MB) may be used.

Recall from Section 9.7 that the ARMv8 architecture provides support for pages and regions of different sizes. Additionally, each TLB entry in the ARMv8 contains a **contiguous bit**. If this bit is set for a particular TLB entry, that entry maps contiguous (adjacent) blocks of memory. Three possible arrangements of contiguous blocks can be mapped in a single TLB entry, thereby increasing the TLB reach:

1. 64-KB TLB entry comprising 16 × 4 KB adjacent blocks.
2. 1-GB TLB entry comprising 32 × 32 MB adjacent blocks.
3. 2-MB TLB entry comprising either 32 × 64 KB adjacent blocks, or 128 × 16 KB adjacent blocks.

Providing support for multiple page sizes may require the operating system—rather than hardware—to manage the TLB. For example, one of the fields in a TLB entry must indicate the size of the page frame corresponding to the entry—or, in the case of ARM architectures, must indicate that the entry refers to a contiguous block of memory. Managing the TLB in software and not hardware comes at a cost in performance. However, the increased hit ratio and TLB reach offset the performance costs.

### 10.9.4 Inverted Page Tables

Section 9.4.3 introduced the concept of the inverted page table. The purpose of this form of page management is to reduce the amount of physical memory needed to track virtual-to-physical address translations. We accomplish this savings by creating a table that has one entry per page of physical memory, indexed by the pair <process-id, page-number>.

Because they keep information about which virtual memory page is stored in each physical frame, inverted page tables reduce the amount of physical memory needed to store this information. However, the inverted page table no longer contains complete information about the logical address space of a process, and that information is required if a referenced page is not currently in memory. Demand paging requires this information to process page faults. For the information to be available, an external page table (one per process) must be kept. Each such table looks like the traditional per-process page table and contains information on where each virtual page is located.

But do external page tables negate the utility of inverted page tables? Since these tables are referenced only when a page fault occurs, they do not need to be available quickly. Instead, they are themselves paged in and out of memory as necessary. Unfortunately, a page fault may now cause the virtual memory manager to generate another page fault as it pages in the external page table it needs to locate the virtual page on the backing store. This special case requires careful handling in the kernel and a delay in the page-lookup processing.

### 10.9.5 Program Structure

Demand paging is designed to be transparent to the user program. In many cases, the user is completely unaware of the paged nature of memory. In other cases, however, system performance can be improved if the user (or compiler) has an awareness of the underlying demand paging.

Let's look at a contrived but informative example. Assume that pages are 128 words in size. Consider a C program whose function is to initialize to 0 each element of a 128-by-128 array. The following code is typical:

```
int i, j;
int[128][128] data;

for (j = 0; j < 128; j++)
    for (i = 0; i < 128; i++)
       data[i][j] = 0;
```

Notice that the array is stored row major; that is, the array is stored `data[0][0]`, `data[0][1]`, · · ·, `data[0][127]`, `data[1][0]`, `data[1][1]`, · · ·, `data[127][127]`. For pages of 128 words, each row takes one page. Thus, the preceding code zeros one word in each page, then another word in each page, and so on. If the operating system allocates fewer than 128 frames to the entire program, then its execution will result in 128 × 128 = 16,384 page faults.

In contrast, suppose we change the code to

```
int i, j;
int[128][128] data;

for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
       data[i][j] = 0;
```

This code zeros all the words on one page before starting the next page, reducing the number of page faults to 128.

Careful selection of data structures and programming structures can increase locality and hence lower the page-fault rate and the number of pages in the working set. For example, a stack has good locality, since access is always made to the top. A hash table, in contrast, is designed to scatter references, producing bad locality. Of course, locality of reference is just one measure of the efficiency of the use of a data structure. Other heavily weighted factors include search speed, total number of memory references, and total number of pages touched.

At a later stage, the compiler and loader can have a significant effect on paging. Separating code and data and generating reentrant code means that code pages can be read-only and hence will never be modified. Clean pages do not have to be paged out to be replaced. The loader can avoid placing routines across page boundaries, keeping each routine completely in one page. Routines that call each other many times can be packed into the same page. This packaging is a variant of the bin-packing problem of operations research: try to pack the variable-sized load segments into the fixed-sized pages so that interpage references are minimized. Such an approach is particularly useful for large page sizes.

### 10.9.6 I/O Interlock and Page Locking

When demand paging is used, we sometimes need to allow some of the pages to be **locked** in memory. One such situation occurs when I/O is done to or from user (virtual) memory. I/O is often implemented by a separate I/O processor. For example, a controller for a USB storage device is generally given the number of bytes to transfer and a memory address for the buffer (Figure 10.28). When the transfer is complete, the CPU is interrupted.

We must be sure the following sequence of events does not occur: A process issues an I/O request and is put in a queue for that I/O device. Meanwhile, the CPU is given to other processes. These processes cause page faults, and one of them, using a global replacement algorithm, replaces the page containing the memory buffer for the waiting process. The pages are paged out. Some time later, when the I/O request advances to the head of the device queue, the I/O occurs to the specified address. However, this frame is now being used for a different page belonging to another process.

There are two common solutions to this problem. One solution is never to execute I/O to user memory. Instead, data are always copied between system memory and user memory. I/O takes place only between system memory and the I/O device. Thus, to write a block on tape, we first copy the block to system memory and then write it to tape. This extra copying may result in unacceptably high overhead.

Another solution is to allow pages to be locked into memory. Here, a lock bit is associated with every frame. If the frame is locked, it cannot be selected for replacement. Under this approach, to write a block to disk, we lock into memory the pages containing the block. The system can then continue as usual. Locked pages cannot be replaced. When the I/O is complete, the pages are unlocked.

Lock bits are used in various situations. Frequently, some or all of the operating-system kernel is locked into memory. Many operating systems cannot tolerate a page fault caused by the kernel or by a specific kernel module, including the one performing memory management. User processes may also need to lock pages into memory. A database process may want to manage a chunk of memory, for example, moving blocks between secondary storage and

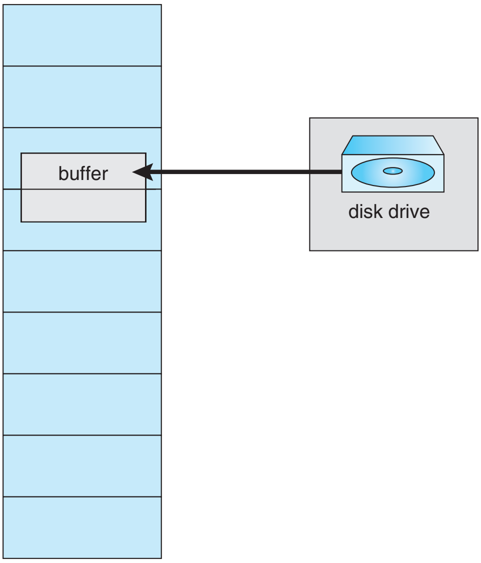


**Figure 10.28** The reason why frames used for I/O must be in memory.

memory itself because it has the best knowledge of how it is going to use its data. Such **pinning** of pages in memory is fairly common, and most operating systems have a system call allowing an application to request that a region of its logical address space be pinned. Note that this feature could be abused and could cause stress on the memory-management algorithms. Therefore, an application frequently requires special privileges to make such a request.

Another use for a lock bit involves normal page replacement. Consider the following sequence of events: A low-priority process faults. Selecting a replacement frame, the paging system reads the necessary page into memory. Ready to continue, the low-priority process enters the ready queue and waits for the CPU. Since it is a low-priority process, it may not be selected by the CPU scheduler for a time. While the low-priority process waits, a high-priority process faults. Looking for a replacement, the paging system sees a page that is in memory but has not been referenced or modified: it is the page that the low-priority process just brought in. This page looks like a perfect replacement. It is clean and will not need to be written out, and it apparently has not been used for a long time.

Whether the high-priority process should be able to replace the low-priority process is a policy decision. After all, we are simply delaying the low-priority process for the benefit of the high-priority process. However, we are wasting the effort spent to bring in the page for the low-priority process. If we decide to prevent replacement of a newly brought-in page until it can be used at least once, then we can use the lock bit to implement this mechanism. When a page is selected for replacement, its lock bit is turned on. It remains on until the faulting process is again dispatched.

Using a lock bit can be dangerous: the lock bit may get turned on but never turned off. Should this situation occur (because of a bug in the operating system, for example), the locked frame becomes unusable. For instance, Solaris allows locking "hints," but it is free to disregard these hints if the free-frame pool becomes too small or if an individual process requests that too many pages be locked in memory.

## 10.10 Operating-System Examples

In this section, we describe how Linux, Windows and Solaris manage virtual memory.

### 10.10.1 Linux

In Section 10.8.2, we discussed how Linux manages kernel memory using slab allocation. We now cover how Linux manages virtual memory. Linux uses demand paging, allocating pages from a list of free frames. In addition, it uses a global page-replacement policy similar to the LRU-approximation clock algorithm described in Section 10.4.5.2. To manage memory, Linux maintains two types of page lists: an `active_list` and an `inactive_list`. The `active_list` contains the pages that are considered in use, while the `inactive_list` contains pages that have not recently been referenced and are eligible to be reclaimed.

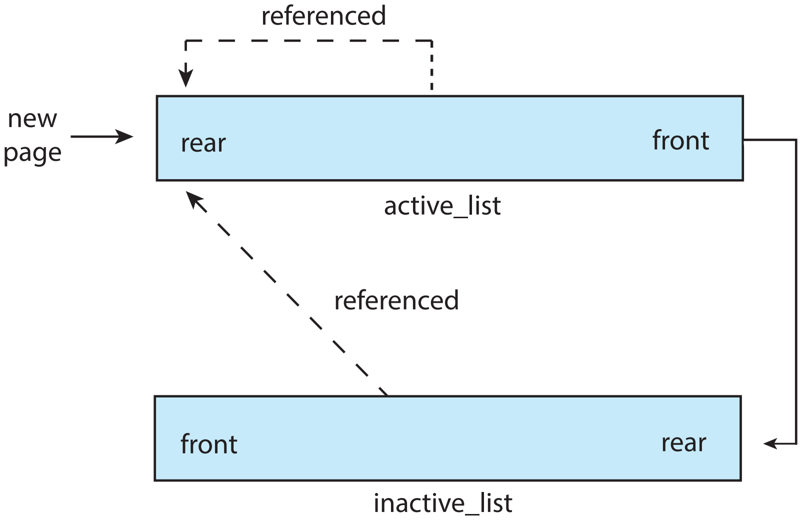


**Figure 10.29** The Linux `active_list` and `inactive_list` structures.

Each page has an ***accessed*** bit that is set whenever the page is referenced. (The actual bits used to mark page access vary by architecture.) When a page is first allocated, its accessed bit is set, and it is added to the rear of the `active_list`. Similarly, whenever a page in the `active_list` is referenced, its accessed bit is set, and the page moves to the rear of the list. Periodically, the accessed bits for pages in the `active_list` are reset. Over time, the least recently used page will be at the front of the `active_list`. From there, it may migrate to the rear of the `inactive_list`. If a page in the `inactive_list` is referenced, it moves back to the rear of the `active_list`. This pattern is illustrated in Figure 10.29.

The two lists are kept in relative balance, and when the `active_list` grows much larger than the `inactive_list`, pages at the front of the `active_list` move to the `inactive_list`, where they become eligible for reclamation. The Linux kernel has a page-out daemon process `kswapd` that periodically awakens and checks the amount of free memory in the system. If free memory falls below a certain threshold, `kswapd` begins scanning pages in the `inactive_list` and reclaiming them for the free list. Linux virtual memory management is discussed in greater detail in Chapter 20.

### 10.10.2 Windows

Windows 10 supports 32- and 64-bit systems running on Intel (IA-32 and x86-64) and ARM architectures. On 32-bit systems, the default virtual address space of a process is 2 GB, although it can be extended to 3 GB. 32-bit systems support 4 GB of physical memory. On 64-bit systems, Windows 10 has a 128-TB virtual address space and supports up to 24 TB of physical memory. (Versions of Windows Server support up to 128 TB of physical memory.) Windows 10 implements most of the memory-management features described thus far, including shared libraries, demand paging, copy-on-write, paging, and memory compression.

Windows 10 implements virtual memory using demand paging with **clustering**, a strategy that recognizes locality of memory references and therefore handles page faults by bringing in not only the faulting page but also several pages immediately preceding and following the faulting page. The size of a cluster varies by page type. For a data page, a cluster contains three pages(the page before and the page after the faulting page); all other page faults have a cluster size of seven.

A key component of virtual memory management in Windows 10 is working-set management. When a process is created, it is assigned a working-set minimum of 50 pages and a working-set maximum of 345 pages. The **working-set minimum** is the minimum number of pages the process is guaranteed to have in memory; if sufficient memory is available, a process may be assigned as many pages as its **working-set maximum**. Unless a process is configured with **hard working-set limits**, these values may be ignored. A process can grow beyond its working-set maximum if sufficient memory is available. Similarly, the amount of memory allocated to a process can shrink below the minimum in periods of high demand for memory.

Windows uses the LRU-approximation clock algorithm, as described in Section 10.4.5.2, with a combination of local and global page-replacement policies. The virtual memory manager maintains a list of free page frames. Associated with this list is a threshold value that indicates whether sufficient free memory is available. If a page fault occurs for a process that is below its working-set maximum, the virtual memory manager allocates a page from the list of free pages. If a process that is at its working-set maximum incurs a page fault and sufficient memory is available, the process is allocated a free page, which allows it to grow beyond its working-set maximum. If the amount of free memory is insufficient, however, the kernel must select a page from the process's working set for replacement using a local LRU page-replacement policy.

When the amount of free memory falls below the threshold, the virtual memory manager uses a global replacement tactic known as **automatic working-set trimming** to restore the value to a level above the threshold. Automatic working-set trimming works by evaluating the number of pages allocated to processes. If a process has been allocated more pages than its working-set minimum, the virtual memory manager removes pages from the working set until either there is sufficient memory available or the process has reached its working-set minimum. Larger processes that have been idle are targeted before smaller, active processes. The trimming procedure continues until there is sufficient free memory, even if it is necessary to remove pages from a process already below its working set minimum. Windows performs working-set trimming on both user-mode and system processes.

### 10.10.3 Solaris

In Solaris, when a thread incurs a page fault, the kernel assigns a page to the faulting thread from the list of free pages it maintains. Therefore, it is imperative that the kernel keep a sufficient amount of free memory available. Associated with this list of free pages is a parameter—`lotsfree`—that represents a threshold to begin paging. The `lotsfree` parameter is typically set to 1/64 the size of the physical memory. Four times per second, the kernel checks whether the amount of free memory is less than `lotsfree`. If the number of free pages falls below `lotsfree`, a process known as a **pageout** starts up. The pageout process is similar to the second-chance algorithm described in Section 10.4.5.2, except that it uses two hands while scanning pages, rather than one.

The pageout process works as follows: The front hand of the clock scans all pages in memory, setting the reference bit to 0. Later, the back hand of the clock examines the reference bit for the pages in memory, appending each page whose reference bit is still set to 0 to the free list and writing its contents to secondary storage if it has been modified. Solaris also manages minor page faults by allowing a process to reclaim a page from the free list if the page is accessed before being reassigned to another process.

The pageout algorithm uses several parameters to control the rate at which pages are scanned (known as the `scanrate`). The scanrate is expressed in pages per second and ranges from `slowscan` to `fastscan`. When free memory falls below `lotsfree`, scanning occurs at `slowscan` pages per second and progresses to `fastscan`, depending on the amount of free memory available. The default value of `slowscan` is 100 pages per second. `Fastscan` is typically set to the value (total physical pages)/2 pages per second, with a maximum of 8,192 pages per second. This is shown in Figure 10.30 (with `fastscan` set to the maximum).

The distance (in pages) between the hands of the clock is determined by a system parameter, `handspread`. The amount of time between the front hand's clearing a bit and the back hand's investigating its value depends on the `scanrate` and the `handspread`. If `scanrate` is 100 pages per second and `handspread` is 1,024 pages, 10 seconds can pass between the time a bit is set by the front hand and the time it is checked by the back hand. However, because of the demands placed on the memory system, a `scanrate` of several thousand is not uncommon. This means that the amount of time between clearing and investigating a bit is often a few seconds.

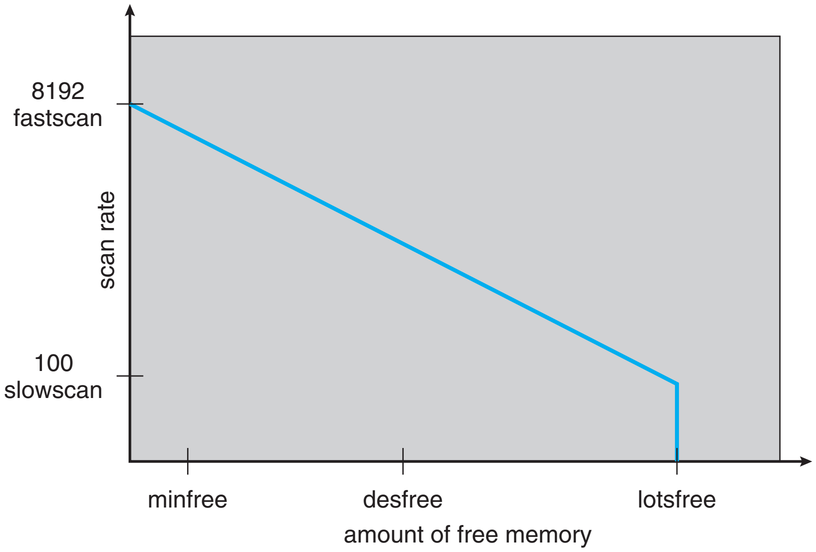


**Figure 10.30** Solaris page scanner.

As mentioned above, the pageout process checks memory four times per second. However, if free memory falls below the value of `desfree` (the desired amount of free memory in the system), pageout will run a hundred times per second with the intention of keeping at least `desfree` free memory available (Figure 10.30). If the pageout process is unable to keep the amount of free memory at `desfree` for a 30-second average, the kernel begins swapping processes, thereby freeing all pages allocated to swapped processes. In general, the kernel looks for processes that have been idle for long periods of time. If the system is unable to maintain the amount of free memory at `minfree`, the pageout process is called for every request for a new page.

The page-scanning algorithm skips pages belonging to libraries that are being shared by several processes, even if they are eligible to be claimed by the scanner. The algorithm also distinguishes between pages allocated to processes and pages allocated to regular data files. This is known as **priority paging** and is covered in Section 14.6.2.

## 10.11 Summary

- Virtual memory abstracts physical memory into an extremely large uniform array of storage.
- The benefits of virtual memory include the following: (1) a program can be larger than physical memory, (2) a program does not need to be entirely in memory, (3) processes can share memory, and (4) processes can be created more efficiently.
- Demand paging is a technique whereby pages are loaded only when they are demanded during program execution. Pages that are never demanded are thus never loaded into memory.
- A page fault occurs when a page that is currently not in memory is accessed. The page must be brought from the backing store into an available page frame in memory.
- Copy-on-write allows a child process to share the same address space as its parent. If either the child or the parent process writes (modifies) a page, a copy of the page is made.
- When available memory runs low, a page-replacement algorithm selects an existing page in memory to replace with a new page. Page-replacement algorithms include FIFO, optimal, and LRU. Pure LRU algorithms are impractical to implement, and most systems instead use LRU-approximation algorithms.
- Global page-replacement algorithms select a page from any process in the system for replacement, while local page-replacement algorithms select a page from the faulting process.
- Thrashing occurs when a system spends more time paging than executing.
- A locality represents a set of pages that are actively used together. As a process executes, it moves from locality to locality. A working set is based on locality and is defined as the set of pages currently in use by a process.

- Memory compression is a memory-management technique that compresses a number of pages into a single page. Compressed memory is an alternative to paging and is used on mobile systems that do not support paging.
- Kernel memory is allocated differently than user-mode processes; it is allocated in contiguous chunks of varying sizes. Two common techniques for allocating kernel memory are (1) the buddy system and (2) slab allocation.
- TLB reach refers to the amount of memory accessible from the TLB and is equal to the number of entries in the TLB multiplied by the page size. One technique for increasing TLB reach is to increase the size of pages.
- Linux, Windows, and Solaris manage virtual memory similarly, using demand paging and copy-on-write, among other features. Each system also uses a variation of LRU approximation known as the clock algorithm.

## Practice Exercises

**10.1** Under what circumstances do page faults occur? Describe the actions taken by the operating system when a page fault occurs.

**10.2** Assume that you have a page-reference string for a process with $m$ frames (initially all empty). The page-reference string has length $p$, and $n$ distinct page numbers occur in it. Answer these questions for any page-replacement algorithms:

a. What is a lower bound on the number of page faults?

b. What is an upper bound on the number of page faults?

**10.3** Consider the following page-replacement algorithms. Rank these algorithms on a five-point scale from “bad” to “perfect” according to their page-fault rate. Separate those algorithms that suffer from Belady’s anomaly from those that do not.

a. LRU replacement

b. FIFO replacement

c. Optimal replacement

d. Second-chance replacement

**10.4** An operating system supports a paged virtual memory. The central processor has a cycle time of 1 microsecond. It costs an additional 1 microsecond to access a page other than the current one. Pages have 1,000 words, and the paging device is a drum that rotates at 3,000 revolutions per minute and transfers 1 million words per second. The following statistical measurements were obtained from the system:

- One percent of all instructions executed accessed a page other than the current page.
- Of the instructions that accessed another page, 80 percent accessed a page already in memory.

- When a new page was required, the replaced page was modified 50 percent of the time.

Calculate the effective instruction time on this system, assuming that the system is running one process only and that the processor is idle during drum transfers.

**10.5** Consider the page table for a system with 12-bit virtual and physical addresses and 256-byte pages.

| Page | Page Frame |
|---|---|
| 0 | – |
| 1 | 2 |
| 2 | C |
| 3 | A |
| 4 | – |
| 5 | 4 |
| 6 | 3 |
| 7 | – |
| 8 | B |
| 9 | 0 |

The list of free page frames is *D*, *E*, *F* (that is, *D* is at the head of the list, *E* is second, and *F* is last). A dash for a page frame indicates that the page is not in memory.

Convert the following virtual addresses to their equivalent physical addresses in hexadecimal. All numbers are given in hexadecimal.

- 9EF
- 111
- 700
- 0FF

**10.6** Discuss the hardware functions required to support demand paging.

**10.7** Consider the two-dimensional array `A`:

```
int A[][] = new int[100][100];
```

where `A[0][0]` is at location 200 in a paged memory system with pages of size 200. A small process that manipulates the matrix resides in page 0 (locations 0 to 199). Thus, every instruction fetch will be from page 0.

For three page frames, how many page faults are generated by the following array-initialization loops? Use LRU replacement, and assume that page frame 1 contains the process and the other two are initially empty.

a.
```
for (int j = 0; j < 100; j++)
    for (int i = 0; i < 100; i++)
        A[i][j] = 0;
```

b.
```
for (int i = 0; i < 100; i++)
    for (int j = 0; j < 100; j++)
        A[i][j] = 0;
```

**10.8** Consider the following page reference string:

1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6.

How many page faults would occur for the following replacement algorithms, assuming one, two, three, four, five, six, and seven frames? Remember that all frames are initially empty, so your first unique pages will cost one fault each.

- LRU replacement
- FIFO replacement
- Optimal replacement

**10.9** Consider the following page reference string:

7, 2, 3, 1, 2, 5, 3, 4, 6, 7, 7, 1, 0, 5, 4, 6, 2, 3, 0 , 1.

Assuming demand paging with three frames, how many page faults would occur for the following replacement algorithms?

- LRU replacement
- FIFO replacement
- Optimal replacement

**10.10** Suppose that you want to use a paging algorithm that requires a reference bit (such as second-chance replacement or working-set model), but the hardware does not provide one. Sketch how you could simulate a reference bit even if one were not provided by the hardware, or explain why it is not possible to do so. If it is possible, calculate what the cost would be.

**10.11** You have devised a new page-replacement algorithm that you think may be optimal. In some contorted test cases, Belady's anomaly occurs. Is the new algorithm optimal? Explain your answer.

**10.12** Segmentation is similar to paging but uses variable-sized "pages." Define two segment-replacement algorithms, one based on the FIFO page-replacement scheme and the other on the LRU page-replacement scheme. Remember that since segments are not the same size, the segment that is chosen for replacement may be too small to leave enough consecutive locations for the needed segment. Consider strategies for systems where segments cannot be relocated and strategies for systems where they can.

**10.13** Consider a demand-paged computer system where the degree of multiprogramming is currently fixed at four. The system was recently measured to determine utilization of the CPU and the paging disk. Three alternative results are shown below. For each case, what is happening? Can the degree of multiprogramming be increased to increase the CPU utilization? Is the paging helping?

a. CPU utilization 13 percent; disk utilization 97 percent

b. CPU utilization 87 percent; disk utilization 3 percent

c. CPU utilization 13 percent; disk utilization 3 percent

**10.14** We have an operating system for a machine that uses base and limit registers, but we have modified the machine to provide a page table. Can the page table be set up to simulate base and limit registers? How can it be, or why can it not be?

## Further Reading

The working-set model was developed by [Denning (1968)]. The enhanced clock algorithm is discussed by [Carr and Hennessy (1981)]. [Russinovich et al. (2017)] describe how Windows implements virtual memory and memory compression. Compressed memory in Windows 10 is further discussed in http://www.makeuseof.com/tag/ram-compression-improves-memory-responsiveness-windows-10.

[McDougall and Mauro (2007)] discuss virtual memory in Solaris. Virtual memory techniques in Linux are described in [Love (2010)] and [Mauerer (2008)]. FreeBSD is described in [McKusick et al. (2015)].

## Bibliography

**[Carr and Hennessy (1981)]** W. R. Carr and J. L. Hennessy, “WSClock—A Simple and Effective Algorithm for Virtual Memory Management”, *Proceedings of the ACM Symposium on Operating Systems Principles* (1981), pages 87–95.

**[Denning (1968)]** P. J. Denning, “The Working Set Model for Program Behavior”, *Communications of the ACM*, Volume 11, Number 5 (1968), pages 323–333.

**[Love (2010)]** R. Love, *Linux Kernel Development*, Third Edition, Developer’s Library (2010).

**[Mauerer (2008)]** W. Mauerer, *Professional Linux Kernel Architecture*, John Wiley and Sons (2008).

**[McDougall and Mauro (2007)]** R. McDougall and J. Mauro, *Solaris Internals*, Second Edition, Prentice Hall (2007).

**[McKusick et al. (2015)]** M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson, *The Design and Implementation of the FreeBSD UNIX Operating System–Second Edition*, Pearson (2015).

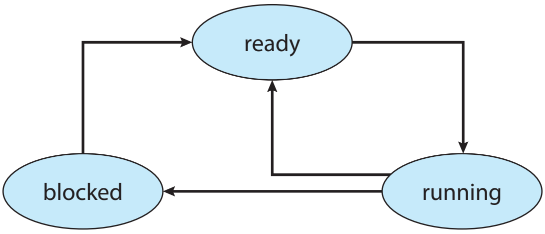
**[Russinovich et al. (2017)]** M. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals–Part 1,* Seventh Edition, Microsoft Press (2017).

## Chapter 10 Exercises

**10.15** Assume that a program has just referenced an address in virtual memory. Describe a scenario in which each of the following can occur. (If no such scenario can occur, explain why.)

- TLB miss with no page fault
- TLB miss with page fault
- TLB hit with no page fault
- TLB hit with page fault

**10.16** A simplified view of thread states is ***ready***, ***running***, and ***blocked***, where a thread is either ready and waiting to be scheduled, is running on the processor, or is blocked (for example, waiting for I/O).



Assuming a thread is in the ***running*** state, answer the following questions, and explain your answers:

a. Will the thread change state if it incurs a page fault? If so, to what state will it change?

b. Will the thread change state if it generates a TLB miss that is resolved in the page table? If so, to what state will it change?

c. Will the thread change state if an address reference is resolved in the page table? If so, to what state will it change?

**10.17** Consider a system that uses pure demand paging.

a. When a process first starts execution, how would you characterize the page-fault rate?

b. Once the working set for a process is loaded into memory, how would you characterize the page-fault rate?

c. Assume that a process changes its locality and the size of the new working set is too large to be stored in available free memory. Identify some options system designers could choose from to handle this situation.

**10.18** The following is a page table for a system with 12-bit virtual and physical addresses and 256-byte pages. Free page frames are to be allocated in the order `9, F, D`. A dash for a page frame indicates that the page is not in memory.

| Page | Page Frame |
|---|---|
| 0 | 0 x 4 |
| 1 | 0 x B |
| 2 | 0 x A |
| 3 | – |
| 4 | – |
| 5 | 0 x 2 |
| 6 | – |
| 7 | 0 x 0 |
| 8 | 0 x C |
| 9 | 0 x 1 |

Convert the following virtual addresses to their equivalent physical addresses in hexadecimal. All numbers are given in hexadecimal. In the case of a page fault, you must use one of the free frames to update the page table and resolve the logical address to its corresponding physical address.

- `0x2A1`
- `0x4E6`
- `0x94A`
- `0x316`

**10.19** What is the copy-on-write feature, and under what circumstances is its use beneficial? What hardware support is required to implement this feature?

**10.20** A certain computer provides its users with a virtual memory space of $2^{32}$ bytes. The computer has $2^{22}$ bytes of physical memory. The virtual memory is implemented by paging, and the page size is 4,096 bytes. A user process generates the virtual address 11123456. Explain how the system establishes the corresponding physical location. Distinguish between software and hardware operations.

**10.21** Assume that we have a demand-paged memory. The page table is held in registers. It takes 8 milliseconds to service a page fault if an empty frame is available or if the replaced page is not modified and 20 milliseconds if the replaced page is modified. Memory-access time is 100 nanoseconds.

Assume that the page to be replaced is modified 70 percent of the time. What is the maximum acceptable page-fault rate for an effective access time of no more than 200 nanoseconds?

**10.22** Consider the page table for a system with 16-bit virtual and physical addresses and 4,096-byte pages.

| Page | Page Frame | Reference Bit |
|---|---|---|
| 0 | 9 | 0 |
| 1 | – | 0 |
| 2 | 10 | 0 |
| 3 | 15 | 0 |
| 4 | 6 | 0 |
| 5 | 13 | 0 |
| 6 | 8 | 0 |
| 7 | 12 | 0 |
| 8 | 7 | 0 |
| 9 | – | 0 |
| 10 | 5 | 0 |
| 11 | 4 | 0 |
| 12 | 1 | 0 |
| 13 | 0 | 0 |
| 14 | – | 0 |
| 15 | 2 | 0 |

The reference bit for a page is set to 1 when the page has been referenced. Periodically, a thread zeroes out all values of the reference bit. A dash for a page frame indicates that the page is not in memory. The page-replacement algorithm is localized LRU, and all numbers are provided in decimal.

a. Convert the following virtual addresses (in hexadecimal) to the equivalent physical addresses. You may provide answers in either hexadecimal or decimal. Also set the reference bit for the appropriate entry in the page table.

- `0x621C`
- `0xF0A3`
- `0xBC1A`
- `0x5BAA`
- `0x0BA1`

b. Using the above addresses as a guide, provide an example of a logical address (in hexadecimal) that results in a page fault.

c. From what set of page frames will the LRU page-replacement algorithm choose in resolving a page fault?

**10.23** When a page fault occurs, the process requesting the page must block while waiting for the page to be brought from disk into physical memory. Assume that there exists a process with five user-level threads and that the mapping of user threads to kernel threads is many to one. If one user thread incurs a page fault while accessing its stack, would the other user threads belonging to the same process also be affected by the page fault—that is, would they also have to wait for the faulting page to be brought into memory? Explain.

**10.24** Apply the (1) FIFO, (2) LRU, and (3) optimal (OPT) replacement algorithms for the following page-reference strings:

- 2, 6, 9, 2, 4, 2, 1, 7, 3, 0, 5, 2, 1, 2, 9, 5, 7, 3, 8, 5
- 0, 6, 3, 0, 2, 6, 3, 5, 2, 4, 1, 3, 0, 6, 1, 4, 2, 3, 5, 7
- 3, 1, 4, 2, 5, 4, 1, 3, 5, 2, 0, 1, 1, 0, 2, 3, 4, 5, 0, 1
- 4, 2, 1, 7, 9, 8, 3, 5, 2, 6, 8, 1, 0, 7, 2, 4, 1, 3, 5, 8
- 0, 1, 2, 3, 4, 4, 3, 2, 1, 0, 0, 1, 2, 3, 4, 4, 3, 2, 1, 0

Indicate the number of page faults for each algorithm assuming demand paging with three frames.

**10.25** Assume that you are monitoring the rate at which the pointer in the clock algorithm moves. (The pointer indicates the candidate page for replacement.) What can you say about the system if you notice the following behavior:

a. Pointer is moving fast.

b. Pointer is moving slow.

**10.26** Discuss situations in which the least frequently used (LFU) page-replacement algorithm generates fewer page faults than the least recently used (LRU) page-replacement algorithm. Also discuss under what circumstances the opposite holds.

**10.27** Discuss situations in which the most frequently used (MFU) page-replacement algorithm generates fewer page faults than the least recently used (LRU) page-replacement algorithm. Also discuss under what circumstances the opposite holds.

**10.28** The KHIE (pronounced “k-hi”) operating system uses a FIFO replacement algorithm for resident pages and a free-frame pool of recently used pages. Assume that the free-frame pool is managed using the LRU replacement policy. Answer the following questions:

a. If a page fault occurs and the page does not exist in the free-frame pool, how is free space generated for the newly requested page?

b. If a page fault occurs and the page exists in the free-frame pool, how are the resident page set and the free-frame pool managed to make space for the requested page?

c. To what does the system degenerate if the number of resident pages is set to one?

d. To what does the system degenerate if the number of pages in the free-frame pool is zero?

**10.29** Consider a demand-paging system with the following time-measured utilizations:

| | |
|---|---|
| CPU utilization | 20% |
| Paging disk | 97.7% |
| Other I/O devices | 5% |

For each of the following, indicate whether it will (or is likely to) improve CPU utilization. Explain your answers.

a. Install a faster CPU.

b. Install a bigger paging disk.

c. Increase the degree of multiprogramming.

d. Decrease the degree of multiprogramming.

e. Install more main memory.

f. Install a faster hard disk or multiple controllers with multiple hard disks.

g. Add prepaging to the page-fetch algorithms.

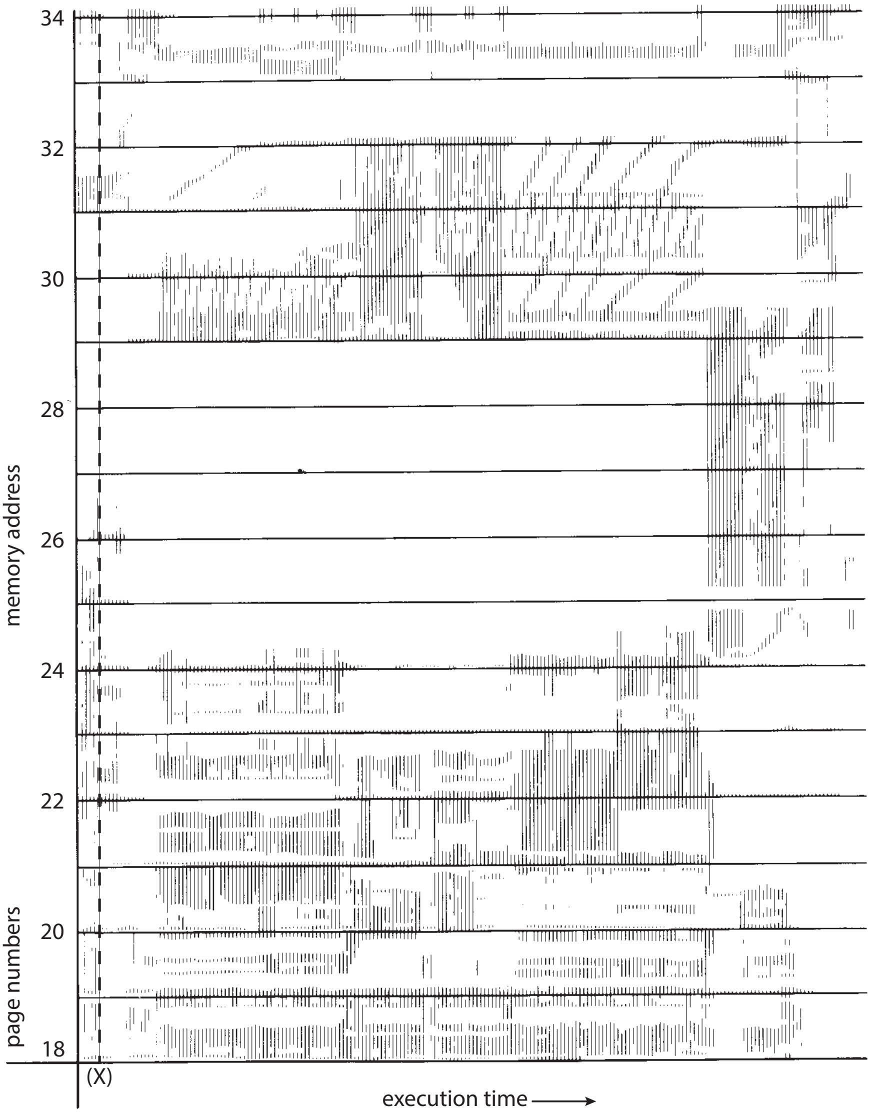
h. Increase the page size.

**10.30** Explain why minor page faults take less time to resolve than major page faults.

**10.31** Explain why compressed memory is used in operating systems for mobile devices.

**10.32** Suppose that a machine provides instructions that can access memory locations using the one-level indirect addressing scheme. What sequence of page faults is incurred when all of the pages of a program are currently nonresident and the first instruction of the program is an indirect memory-load operation? What happens when the operating system is using a per-process frame allocation technique and only two pages are allocated to this process?

**10.33** Consider the page references:



What pages represent the locality at time (*X*)?

**10.34** Suppose that your replacement policy (in a paged system) is to examine each page regularly and to discard that page if it has not been used since the last examination. What would you gain and what would you lose by using this policy rather than LRU or second-chance replacement?

**10.35** A page-replacement algorithm should minimize the number of page faults. We can achieve this minimization by distributing heavily used pages evenly over all of memory, rather than having them compete for a small number of page frames. We can associate with each page frame a counter of the number of pages associated with that frame. Then, to replace a page, we can search for the page frame with the smallest counter.

a. Define a page-replacement algorithm using this basic idea. Specifically address these problems:

   - What is the initial value of the counters?
   - When are counters increased?
   - When are counters decreased?
   - How is the page to be replaced selected?

b. How many page faults occur for your algorithm for the following reference string with four page frames?

   1, 2, 3, 4, 5, 3, 4, 1, 6, 7, 8, 7, 8, 9, 7, 8, 9, 5, 4, 5, 4, 2.

c. What is the minimum number of page faults for an optimal page-replacement strategy for the reference string in part b with four page frames?

**10.36** Consider a demand-paging system with a paging disk that has an average access and transfer time of 20 milliseconds. Addresses are translated through a page table in main memory, with an access time of 1 microsecond per memory access. Thus, each memory reference through the page table takes two accesses. To improve this time, we have added an associative memory that reduces access time to one memory reference if the page-table entry is in the associative memory.

Assume that 80 percent of the accesses are in the associative memory and that, of those remaining, 10 percent (or 2 percent of the total) cause page faults. What is the effective memory access time?

**10.37** What is the cause of thrashing? How does the system detect thrashing? Once it detects thrashing, what can the system do to eliminate this problem?

**10.38** Is it possible for a process to have two working sets, one representing data and another representing code? Explain.

**10.39** Consider the parameter $\Delta$ used to define the working-set window in the working-set model. When $\Delta$ is set to a low value, what is the effect on the page-fault frequency and the number of active (nonsuspended) processes currently executing in the system? What is the effect when $\Delta$ is set to a very high value?

**10.40** In a 1,024-KB segment, memory is allocated using the buddy system. Using Figure 10.26 as a guide, draw a tree illustrating how the following memory requests are allocated:

- Request 5-KB
- Request 135 KB.
- Request 14 KB.
- Request 3 KB.

- Request 12 KB.

Next, modify the tree for the following releases of memory. Perform coalescing whenever possible:

- Release 3 KB.
- Release 5 KB.
- Release 14 KB.
- Release 12 KB.

**10.41** A system provides support for user-level and kernel-level threads. The mapping in this system is one to one (there is a corresponding kernel thread for each user thread). Does a multithreaded process consist of (a) a working set for the entire process or (b) a working set for each thread? Explain

**10.42** The slab-allocation algorithm uses a separate cache for each different object type. Assuming there is one cache per object type, explain why this scheme doesn't scale well with multiple CPUs. What could be done to address this scalability issue?

**10.43** Consider a system that allocates pages of different sizes to its processes. What are the advantages of such a paging scheme? What modifications to the virtual memory system would be needed to provide this functionality?

# Programming Problems

**10.44** Write a program that implements the FIFO, LRU, and optimal (OPT) page-replacement algorithms presented in Section 10.4. Have your program initially generate a random page-reference string where page numbers range from 0 to 9. Apply the random page-reference string to each algorithm, and record the number of page faults incurred by each algorithm. Pass the number of page frames to the program at startup. You may implement this program in any programming language of your choice. (You may find your implementation of either FIFO or LRU to be helpful in the virtual memory manager programming project.)

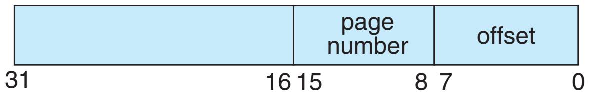
# Programming Projects

## Designing a Virtual Memory Manager

This project consists of writing a program that translates logical to physical addresses for a virtual address space of size $2^{16} = 65{,}536$ bytes. Your program will read from a file containing logical addresses and, using a TLB and a page table, will translate each logical address to its corresponding physical address and output the value of the byte stored at the translated physical address. Your learning goal is to use simulation to understand the steps involved in translating logical to physical addresses. This will include resolving page faults using demand paging, managing a TLB, and implementing a page-replacement algorithm.

### Specific

Your program will read a file containing several 32-bit integer numbers that represent logical addresses. However, you need only be concerned with 16-bit addresses, so you must mask the rightmost 16 bits of each logical address. These 16 bits are divided into (1) an 8-bit page number and (2) an 8-bit page offset. Hence, the addresses are structured as shown as:



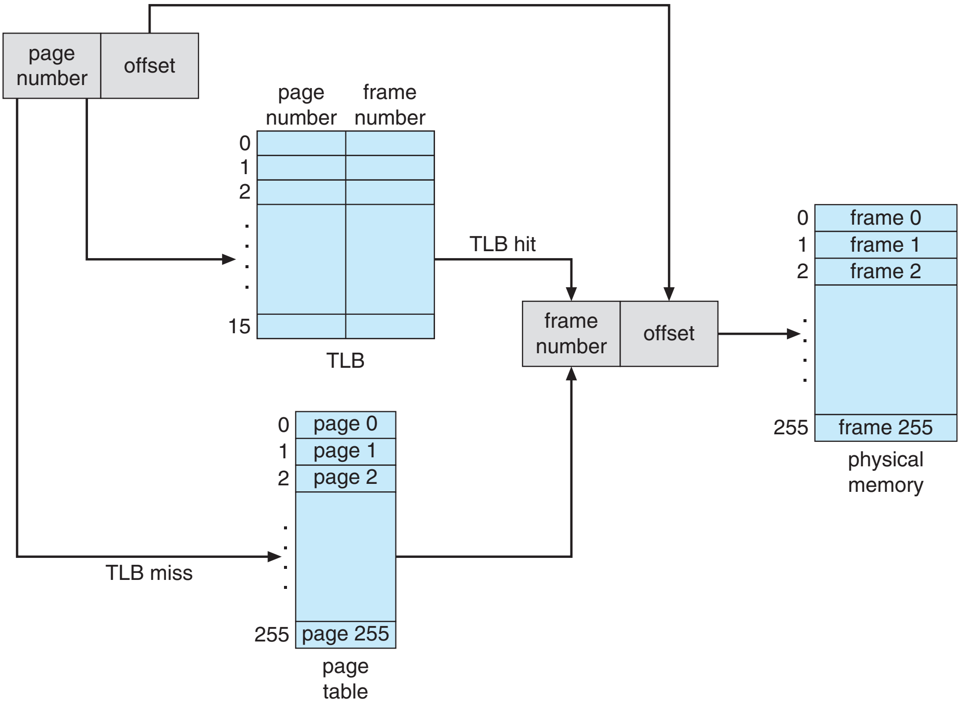
Other specifics include the following:

- $2^8$ entries in the page table
- Page size of $2^8$ bytes
- 16 entries in the TLB
- Frame size of $2^8$ bytes
- 256 frames
- Physical memory of 65,536 bytes (256 frames × 256-byte frame size)

Additionally, your program need only be concerned with reading logical addresses and translating them to their corresponding physical addresses. You do not need to support writing to the logical address space.

### Address Translation

Your program will translate logical to physical addresses using a TLB and page table as outlined in Section 9.3. First, the page number is extracted from the logical address, and the TLB is consulted. In the case of a TLB hit, the frame number is obtained from the TLB. In the case of a TLB miss, the page table must be consulted. In the latter case, either the frame number is obtained from the page table, or a page fault occurs. A visual representation of the address-translation process is:


page number
offset
page number
frame number
0
1
2
15
TLB
TLB hit
frame number
offset
0
1
2
255
frame 0
frame 1
frame 2
frame 255
physical memory
0
1
2
255
page 0
page 1
page 2
page 255
page table
TLB miss

### Handling Page Faults

Your program will implement demand paging as described in Section 10.2. The backing store is represented by the file `BACKING_STORE.bin`, a binary file of size 65,536 bytes. When a page fault occurs, you will read in a 256-byte page from the file `BACKING_STORE` and store it in an available page frame in physical memory. For example, if a logical address with page number 15 resulted in a page fault, your program would read in page 15 from `BACKING_STORE` (remember that pages begin at 0 and are 256 bytes in size) and store it in a page frame in physical memory. Once this frame is stored (and the page table and TLB are updated), subsequent accesses to page 15 will be resolved by either the TLB or the page table.

You will need to treat `BACKING_STORE.bin` as a random-access file so that you can randomly seek to certain positions of the file for reading. We suggest using the standard C library functions for performing I/O, including `fopen()`, `fread()`, `fseek()`, and `fclose()`.

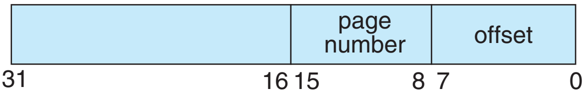
The size of physical memory is the same as the size of the virtual address space—65,536 bytes—so you do not need to be concerned about page replacements during a page fault. Later, we describe a modification to this project using a smaller amount of physical memory; at that point, a page-replacement strategy will be required.

### Test File

We provide the file `addresses.txt`, which contains integer values representing logical addresses ranging from $0 to 65535$ (the size of the virtual address space). Your program will open this file, read each logical address and translate it to its corresponding physical address, and output the value of the signed byte at the physical address.

### How to Begin

First, write a simple program that extracts the page number and offset based on:



from the following integer numbers:

$1, 256, 32768, 32769, 128, 65534, 33153$

Perhaps the easiest way to do this is by using the operators for bit-masking and bit-shifting. Once you can correctly establish the page number and offset from an integer number, you are ready to begin.

Initially, we suggest that you bypass the TLB and use only a page table. You can integrate the TLB once your page table is working properly. Remember, address translation can work without a TLB; the TLB just makes it faster. When you are ready to implement the TLB, recall that it has only sixteen entries, so you will need to use a replacement strategy when you update a full TLB. You may use either a FIFO or an LRU policy for updating your TLB.

### How to Run Your Program

Your program should run as follows:

```
./a.out addresses.txt
```

Your program will read in the file `addresses.txt`, which contains 1,000 logical addresses ranging from 0 to 65535. Your program is to translate each logical address to a physical address and determine the contents of the signed byte stored at the correct physical address. (Recall that in the C language, the `char` data type occupies a byte of storage, so we suggest using `char` values.)

Your program is to output the following values:

1. The logical address being translated (the integer value being read from `addresses.txt`).
2. The corresponding physical address (what your program translates the logical address to).
3. The signed byte value stored in physical memory at the translated physical address.

We also provide the file `correct.txt`, which contains the correct output values for the file `addresses.txt`. You should use this file to determine if your program is correctly translating logical to physical addresses.

### Statistics

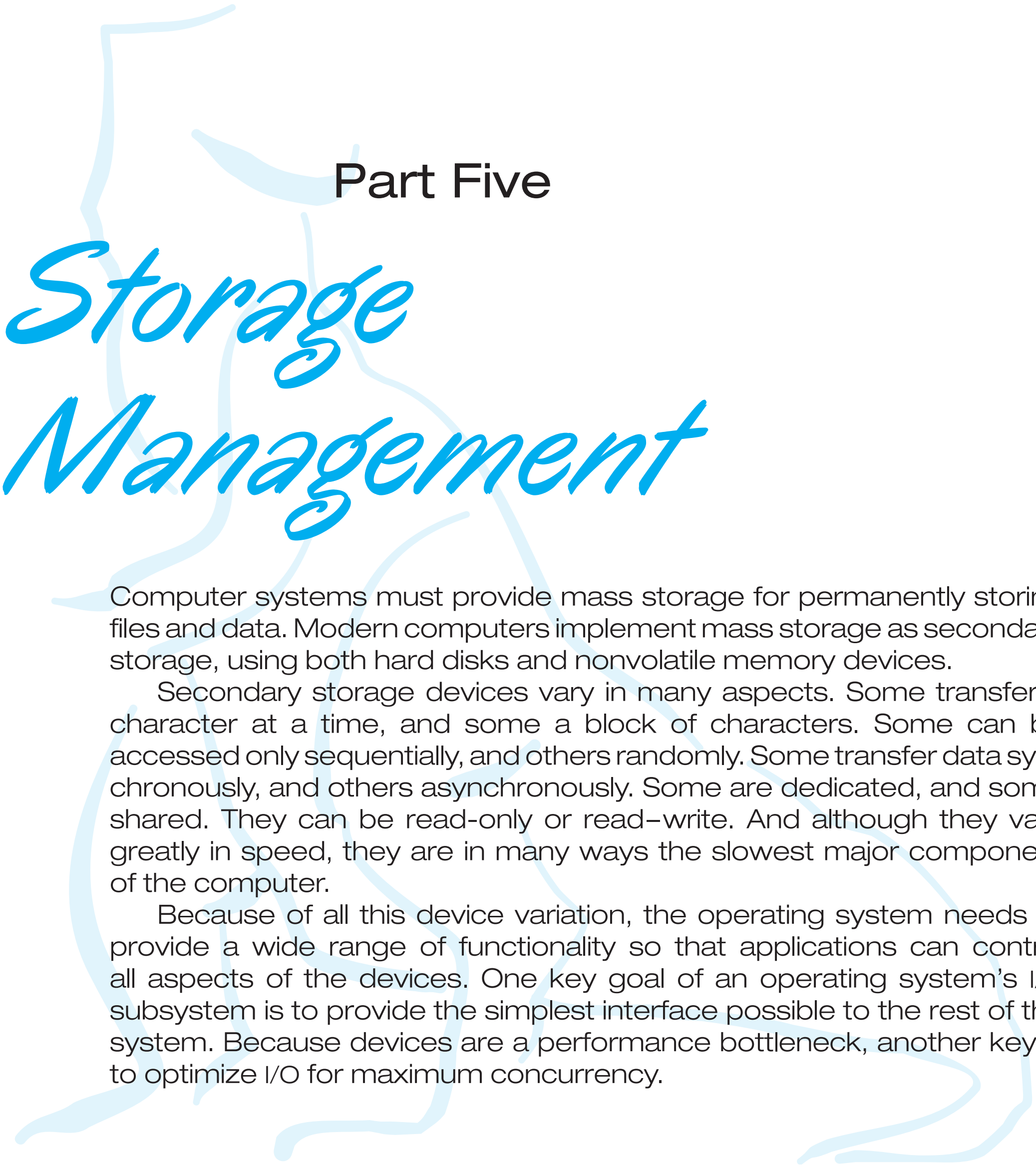
After completion, your program is to report the following statistics:

1. Page-fault rate—The percentage of address references that resulted in page faults.
2. TLB hit rate—The percentage of address references that were resolved in the TLB.

Since the logical addresses in `addresses.txt` were generated randomly and do not reflect any memory access locality, do not expect to have a high TLB hit rate.

### Page Replacement

Thus far, this project has assumed that physical memory is the same size as the virtual address space. In practice, physical memory is typically much smaller than a virtual address space. This phase of the project now assumes using a smaller physical address space with 128 page frames rather than 256. This change will require modifying your program so that it keeps track of free page frames as well as implementing a page-replacement policy using either FIFO or LRU (Section 10.4) to resolve page faults when there is no free memory.



## Part Five

# Storage Management

Computer systems must provide mass storage for permanently storing files and data. Modern computers implement mass storage as secondary storage, using both hard disks and nonvolatile memory devices.

Secondary storage devices vary in many aspects. Some transfer a character at a time, and some a block of characters. Some can be accessed only sequentially, and others randomly. Some transfer data synchronously, and others asynchronously. Some are dedicated, and some shared. They can be read-only or read–write. And although they vary greatly in speed, they are in many ways the slowest major component of the computer.

Because of all this device variation, the operating system needs to provide a wide range of functionality so that applications can control all aspects of the devices. One key goal of an operating system's I/O subsystem is to provide the simplest interface possible to the rest of the system. Because devices are a performance bottleneck, another key is to optimize I/O for maximum concurrency.

# CHAPTER 11

# Mass-Storage Structure

In this chapter, we discuss how mass storage—the nonvolatile storage system of a computer—is structured. The main mass-storage system in modern computers is secondary storage, which is usually provided by hard disk drives (HDD) and nonvolatile memory (NVM) devices. Some systems also have slower, larger, tertiary storage, generally consisting of magnetic tape, optical disks, or even cloud storage.

Because the most common and important storage devices in modern computer systems are HDDs and NVM devices, the bulk of this chapter is devoted to discussing these two types of storage. We first describe their physical structure. We then consider scheduling algorithms, which schedule the order of I/Os to maximize performance. Next, we discuss device formatting and management of boot blocks, damaged blocks, and swap space. Finally, we examine the structure of RAID systems.

There are many types of mass storage, and we use the general term ***nonvolatile storage*** (NVS) or talk about storage "drives" when the discussion includes all types. Particular devices, such as HDDs and NVM devices, are specified as appropriate.

## CHAPTER OBJECTIVES

- Describe the physical structures of various secondary storage devices and the effect of a device's structure on its uses.
- Explain the performance characteristics of mass-storage devices.
- Evaluate I/O scheduling algorithms.
- Discuss operating-system services provided for mass storage, including RAID.

## 11.1 Overview of Mass-Storage Structure

The bulk of secondary storage for modern computers is provided by **hard disk drives** (**HDDs**) and **nonvolatile memory** (**NVM**) devices. In this section,

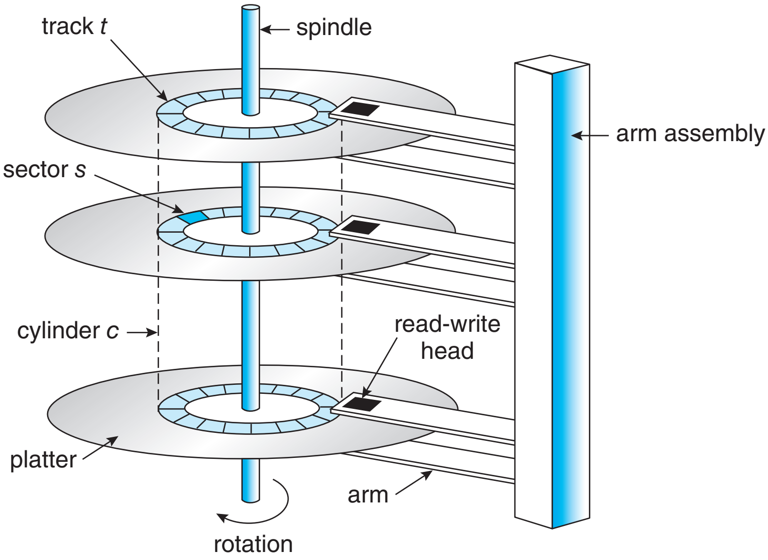


**Figure 11.1** HDD moving-head disk mechanism.

we describe the basic mechanisms of these devices and explain how operating systems translate their physical properties to logical storage via address mapping.

### 11.1.1 Hard Disk Drives

Conceptually, HDDs are relatively simple (Figure 11.1). Each disk **platter** has a flat circular shape, like a CD. Common platter diameters range from 1.8 to 3.5 inches. The two surfaces of a platter are covered with a magnetic material. We store information by recording it magnetically on the platters, and we read information by detecting the magnetic pattern on the platters.

A read–write head "flies" just above each surface of every platter. The heads are attached to a **disk arm** that moves all the heads as a unit. The surface of a platter is logically divided into circular **tracks**, which are subdivided into **sectors**. The set of tracks at a given arm position make up a **cylinder**. There may be thousands of concentric cylinders in a disk drive, and each track may contain hundreds of sectors. Each sector has a fixed size and is the smallest unit of transfer. The sector size was commonly 512 bytes until around 2010. At that point, many manufacturers start migrating to 4KB sectors. The storage capacity of common disk drives is measured in gigabytes and terabytes. A disk drive with the cover removed is shown in Figure 11.2.

A disk drive motor spins it at high speed. Most drives rotate 60 to 250 times per second, specified in terms of rotations per minute (**RPM**). Common drives spin at 5,400, 7,200, 10,000, and 15,000 RPM. Some drives power down when not in use and spin up upon receiving an I/O request. Rotation speed relates to transfer rates. The **transfer rate** is the rate at which data flow between the drive and the computer. Another performance aspect, the **positioning time**, or **random-access time**, consists of two parts: the time necessary to move the disk arm to the desired cylinder, called the **seek time**, and the time necessary for the

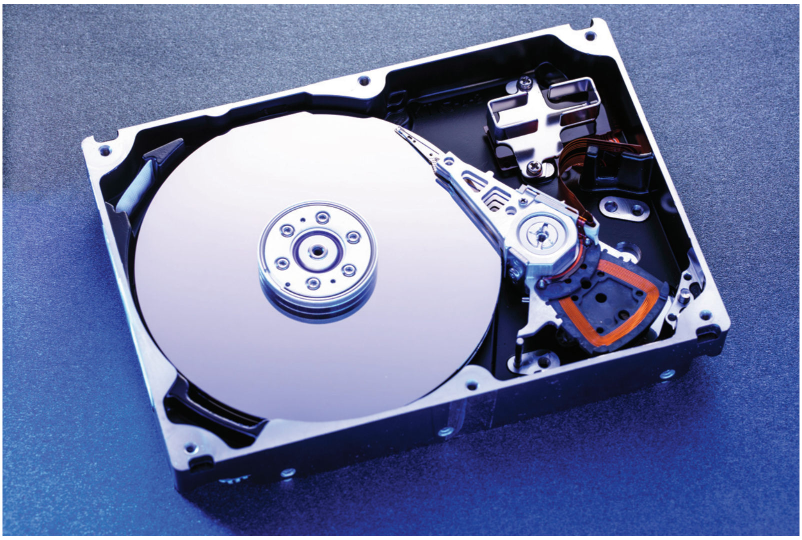


**Figure 11.2** A 3.5-inch HDD with cover removed.

desired sector to rotate to the disk head, called the **rotational latency**. Typical disks can transfer tens to hundreds of megabytes of data per second, and they have seek times and rotational latencies of several milliseconds. They increase performance by having DRAM buffers in the drive controller.

The disk head flies on an extremely thin cushion (measured in microns) of air or another gas, such as helium, and there is a danger that the head will make contact with the disk surface. Although the disk platters are coated with a thin protective layer, the head will sometimes damage the magnetic surface. This accident is called a **head crash**. A head crash normally cannot be repaired; the entire disk must be replaced, and the data on the disk are lost unless they were backed up to other storage or RAID protected. (RAID is discussed in Section 11.8.)

HDDs are sealed units, and some chassis that hold HDDs allow their removal without shutting down the system or storage chassis. This is helpful when a system needs more storage than can be connected at a given time or when it is necessary to replace a bad drive with a working one. Other types of storage media are also **removable**, including CDs, DVDs, and Blu-ray discs.

### *DISK TRANSFER RATES*

As with many aspects of computing, published performance numbers for disks are not the same as real-world performance numbers. Stated transfer rates are always higher than **effective transfer rates**, for example. The transfer rate may be the rate at which bits can be read from the magnetic media by the disk head, but that is different from the rate at which blocks are delivered to the operating system.

### 11.1.2 Nonvolatile Memory Devices

Nonvolatile memory (NVM) devices are growing in importance. Simply described, NVM devices are electrical rather than mechanical. Most commonly, such a device is composed of a controller and flash NAND die semiconductor chips, which are used to store data. Other NVM technologies exist, like DRAM with battery backing so it doesn't lose its contents, as well as other semiconductor technology like 3D XPoint, but they are far less common and so are not discussed in this book.

#### 11.1.2.1 Overview of Nonvolatile Memory Devices

Flash-memory-based NVM is frequently used in a disk-drive-like container, in which case it is called a **solid-state disk** (**SSD**) (Figure 11.3). In other instances, it takes the form of a **USB drive** (also known as a thumb drive or flash drive) or a DRAM stick. It is also surface-mounted onto motherboards as the main storage in devices like smartphones. In all forms, it acts and can be treated in the same way. Our discussion of NVM devices focuses on this technology.

NVM devices can be more reliable than HDDs because they have no moving parts and can be faster because they have no seek time or rotational latency. In addition, they consume less power. On the negative side, they are more expensive per megabyte than traditional hard disks and have less capacity than the larger hard disks. Over time, however, the capacity of NVM devices has increased faster than HDD capacity, and their price has dropped more quickly, so their use is increasing dramatically. In fact, SSDs and similar devices are now used in some laptop computers to make them smaller, faster, and more energy-efficient.

Because NVM devices can be much faster than hard disk drives, standard bus interfaces can cause a major limit on throughput. Some NVM devices are designed to connect directly to the system bus (PCIe, for example). This technology is changing other traditional aspects of computer design as well.

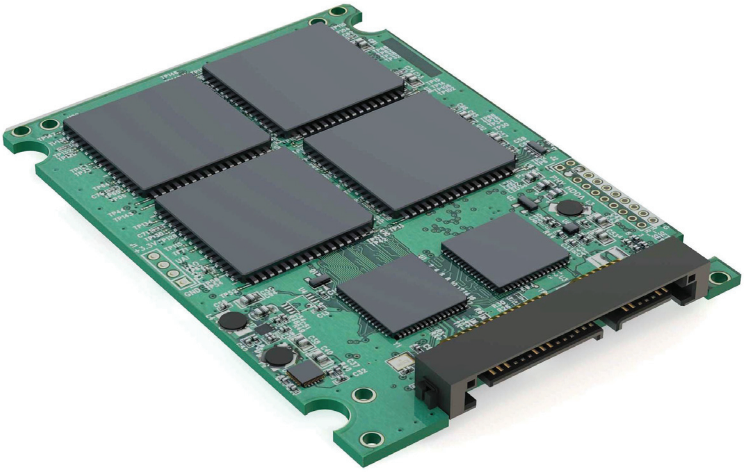


**Figure 11.3** A 3.5-inch SSD circuit board.

Some systems use it as a direct replacement for disk drives, while others use it as a new cache tier, moving data among magnetic disks, NVM, and main memory to optimize performance.

NAND semiconductors have some characteristics that present their own storage and reliability challenges. For example, they can be read and written in a "page" increment (similar to a sector), but data cannot be overwritten—rather, the NAND cells have to be erased first. The erasure, which occurs in a "block" increment that is several pages in size, takes much more time than a read (the fastest operation) or a write (slower than read, but much faster than erase). Helping the situation is that NVM flash devices are composed of many die, with many datapaths to each die, so operations can happen in parallel (each using a datapath). NAND semiconductors also deteriorate with every erase cycle, and after approximately 100,000 program-erase cycles (the specific number varies depending on the medium), the cells no longer retain data. Because of the write wear, and because there are no moving parts, NAND NVM lifespan is not measured in years but in **Drive Writes Per Day** (**DWPD**). That measure is how many times the drive capacity can be written per day before the drive fails. For example, a 1 TB NAND drive with a 5 DWPD rating is expected to have 5 TB per day written to it for the warranty period without failure.

These limitations have led to several ameliorating algorithms. Fortunately, they are usually implemented in the NVM device controller and are not of concern to the operating system. The operating system simply reads and writes logical blocks, and the device manages how that is done. (Logical blocks are discussed in more detail in Section 11.1.5.) However, NVM devices have performance variations based on their operating algorithms, so a brief discussion of what the controller does is warranted.

#### 11.1.2.2 NAND Flash Controller Algorithms

Because NAND semiconductors cannot be overwritten once written, there are usually pages containing invalid data. Consider a file-system block, written once and then later written again. If no erase has occurred in the meantime, the page first written has the old data, which are now invalid, and the second page has the current, good version of the block. A NAND block containing valid and invalid pages is shown in Figure 11.4. To track which logical blocks contain valid data, the controller maintains a **flash translation layer** (**FTL**). This table maps which physical pages contain currently valid logical blocks. It also

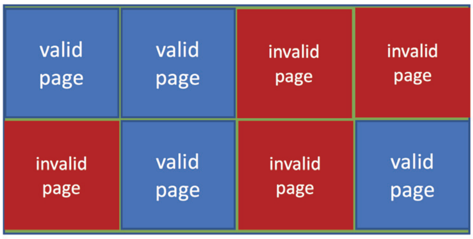


**Figure 11.4** A NAND block with valid and invalid pages.

tracks physical block state—that is, which blocks contain only invalid pages and therefore can be erased.

Now consider a full SSD with a pending write request. Because the SSD is full, all pages have been written to, but there might be a block that contains no valid data. In that case, the write could wait for the erase to occur, and then the write could occur. But what if there are no free blocks? There still could be some space available if individual pages are holding invalid data. In that case, **garbage collection** could occur—good data could be copied to other locations, freeing up blocks that could be erased and could then receive the writes. However, where would the garbage collection store the good data? To solve this problem and improve write performance, the NVM device uses **over-provisioning**. The device sets aside a number of pages (frequently 20 percent of the total) as an area always available to write to. Blocks that are totally invalid by garbage collection, or write operations invalidating older versions of the data, are erased and placed in the over-provisioning space if the device is full or returned to the free pool.

The over-provisioning space can also help with **wear leveling**. If some blocks are erased repeatedly, while others are not, the frequently erased blocks will wear out faster than the others, and the entire device will have a shorter lifespan than it would if all the blocks wore out concurrently. The controller tries to avoid that by using various algorithms to place data on less-erased blocks so that subsequent erases will happen on those blocks rather than on the more erased blocks, leveling the wear across the entire device.

In terms of data protection, like HDDs, NVM devices provide error-correcting codes, which are calculated and stored along with the data during writing and read with the data to detect errors and correct them if possible. (Error-correcting codes are discussed in Section 11.5.1.) If a page frequently has correctible errors, the page might be marked as bad and not used in subsequent writes. Generally, a single NVM device, like an HDD, can have a catastrophic failure in which it corrupts or fails to reply to read or write requests. To allow data to be recoverable in those instances, RAID protection is used.

### 11.1.3 Volatile Memory

It might seem odd to discuss volatile memory in a chapter on mass-storage structure, but it is justifiable because DRAM is frequently used as a mass-storage device. Specifically, **RAM drives** (which are known by many names, including RAM disks) act like secondary storage but are created by device drivers that carve out a section of the system's DRAM and present it to the rest of the system as it if were a storage device. These "drives" can be used as raw block devices, but more commonly, file systems are created on them for standard file operations.

Computers already have buffering and caching, so what is the purpose of yet another use of DRAM for temporary data storage? After all, DRAM is volatile, and data on a RAM drive does not survive a system crash, shutdown, or power down. Caches and buffers are allocated by the programmer or operating system, whereas RAM drives allow the user (as well as the programmer) to place

### *MAGNETIC TAPES*

**Magnetic tape** was used as an early secondary-storage medium. Although it is nonvolatile and can hold large quantities of data, its access time is slow compared with that of main memory and drives. In addition, random access to magnetic tape is about a thousand times slower than random access to HDDs and about a hundred thousand times slower than random access to SSDs so tapes are not very useful for secondary storage. Tapes are used mainly for backup, for storage of infrequently used information, and as a medium for transferring information from one system to another.

A tape is kept in a spool and is wound or rewound past a read–write head. Moving to the correct spot on a tape can take minutes, but once positioned, tape drives can read and write data at speeds comparable to HDDs. Tape capacities vary greatly, depending on the particular kind of tape drive, with current capacities exceeding several terabytes. Some tapes have built-in compression that can more than double the effective storage. Tapes and their drivers are usually categorized by width, including 4, 8, and 19 millimeters and 1/4 and 1/2 inch. Some are named according to technology, such as LTO-6 (Figure 11.5) and SDLT.

**Figure 11.5** An LTO-6 Tape drive with tape cartridge inserted.

data in memory for temporary safekeeping using standard file operations. In fact, RAM drive functionality is useful enough that such drives are found in all major operating systems. On Linux there is `/dev/ram`, on macOS the `diskutil` command creates them, Windows has them via third-party tools, and Solaris and Linux create `/tmp` at boot time of type “tmpfs”, which is a RAM drive.

RAM drives are useful as high-speed temporary storage space. Although NVM devices are fast, DRAM is much faster, and I/O operations to RAM drives are the fastest way to create, read, write, and delete files and their contents. Many programs use (or could benefit from using) RAM drives for storing temporary files. For example, programs can share data easily by writing and reading files from a RAM drive. For another example, Linux at boot time creates a temporary root file system (`initrd`) that allows other parts of the system to have access to a root file system and its contents before the parts of the operating system that understand storage devices are loaded.

### 11.1.4 Secondary Storage Connection Methods

A secondary storage device is attached to a computer by the system bus or an **I/O bus**. Several kinds of buses are available, including **advanced technology attachment** (**ATA**), **serial ATA** (**SATA**), **eSATA**, **serial attached SCSI** (**SAS**), **universal serial bus** (**USB**), and **fibr channel** (**FC**). The most common connection method is SATA. Because NVM devices are much faster than HDDs, the industry created a special, fast interface for NVM devices called **NVM express** (**NVMe**). NVMe directly connects the device to the system PCI bus, increasing throughput and decreasing latency compared with other connection methods.

The data transfers on a bus are carried out by special electronic processors called **controllers** (or **host-bus adapters** (**HBA**)). The **host controller** is the controller at the computer end of the bus. A **device controller** is built into each storage device. To perform a mass storage I/O operation, the computer places a command into the host controller, typically using memory-mapped I/O ports, as described in Section 12.2.1. The host controller then sends the command via messages to the device controller, and the controller operates the drive hardware to carry out the command. Device controllers usually have a built-in cache. Data transfer at the drive happens between the cache and the storage media, and data transfer to the host, at fast electronic speeds, occurs between the cache host DRAM via DMA.

### 11.1.5 Address Mapping

Storage devices are addressed as large one-dimensional arrays of **logical blocks**, where the logical block is the smallest unit of transfer. Each logical block maps to a physical sector or semiconductor page. The one-dimensional array of logical blocks is mapped onto the sectors or pages of the device. Sector 0 could be the first sector of the first track on the outermost cylinder on an HDD, for example. The mapping proceeds in order through that track, then through the rest of the tracks on that cylinder, and then through the rest of the cylinders, from outermost to innermost. For NVM the mapping is from a tuple (finite ordered list) of chip, block, and page to an array of logical blocks. A logical block address (**LBA**) is easier for algorithms to use than a sector, cylinder, head tuple or chip, block, page tuple.

By using this mapping on an HDD, we can—at least in theory—convert a logical block number into an old-style disk address that consists of a cylinder number, a track number within that cylinder, and a sector number within that track. In practice, it is difficult to perform this translation, for three reasons. First, most drives have some defective sectors, but the mapping hides this by substituting spare sectors from elsewhere on the drive. The logical block address stays sequential, but the physical sector location changes. Second, the number of sectors per track is not a constant on some drives. Third, disk manufacturers manage LBA to physical address mapping internally, so in current drives there is little relationship between LBA and physical sectors. In spite of these physical address vagaries, algorithms that deal with HDDs tend to assume that logical addresses are relatively related to physical addresses. That is, ascending logical addresses tend to mean ascending physical address.

Let's look more closely at the second reason. On media that use **constant linear velocity** (**CLV**), the density of bits per track is uniform. The farther a track is from the center of the disk, the greater its length, so the more sectors it can hold. As we move from outer zones to inner zones, the number of sectors per track decreases. Tracks in the outermost zone typically hold 40 percent more sectors than do tracks in the innermost zone. The drive increases its rotation speed as the head moves from the outer to the inner tracks to keep the same rate of data moving under the head. This method is used in CD-ROM and DVD-ROM drives. Alternatively, the disk rotation speed can stay constant; in this case, the density of bits decreases from inner tracks to outer tracks to keep the data rate constant (and performance relatively the same no matter where data is on the drive). This method is used in hard disks and is known as **constant angular velocity** (**CAV**).

The number of sectors per track has been increasing as disk technology improves, and the outer zone of a disk usually has several hundred sectors per track. Similarly, the number of cylinders per disk has been increasing; large disks have tens of thousands of cylinders.

Note that there are more types of storage devices than are reasonable to cover in an operating systems text. For example, there are "shingled magnetic recording" hard drives with higher density but worse performance than mainstream HDDs (see http://www.tomsitpro.com/articles/shingled-magnetic-recoding-smr-101-basics,2-933.html). There are also combination devices that include NVM and HDD technology, or volume managers (see Section 11.5) that can knit together NVM and HDD devices into a storage unit faster than HDD but lower cost than NVM. These devices have different characteristics from the more common devices, and might need different caching and scheduling algorithms to maximize performance.

## 11.2 HDD Scheduling

One of the responsibilities of the operating system is to use the hardware efficiently. For HDDs, meeting this responsibility entails minimizing access time and maximizing data transfer bandwidth.

For HDDs and other mechanical storage devices that use platters, access time has two major components, as mentioned in Section 11.1. The seek time is the time for the device arm to move the heads to the cylinder containing the desired sector, and the rotational latency is the additional time for the platter to rotate the desired sector to the head. The device **bandwidth** is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer. We can improve both the access time and the bandwidth by managing the order in which storage I/O requests are serviced.

Whenever a process needs I/O to or from the drive, it issues a system call to the operating system. The request specifies several pieces of information:

- Whether this operation is input or output
- The open file handle indicating the file to operate on
- What the memory address for the transfer is
- The amount of data to transfer

If the desired drive and controller are available, the request can be serviced immediately. If the drive or controller is busy, any new requests for service will be placed in the queue of pending requests for that drive. For a multiprogramming system with many processes, the device queue may often have several pending requests.

The existence of a queue of requests to a device that can have its performance optimized by avoiding head seeks allows device drivers a chance to improve performance via queue ordering.

In the past, HDD interfaces required that the host specify which track and which head to use, and much effort was spent on disk scheduling algorithms. Drives newer than the turn of the century not only do not expose these controls to the host, but also map LBA to physical addresses under drive control. The current goals of disk scheduling include fairness, timeliness, and optimizations, such as bunching reads or writes that appear in sequence, as drives perform best with sequential I/O. Therefore some scheduling effort is still useful. Any one of several disk-scheduling algorithms can be used, and we discuss them next. Note that absolute knowledge of head location and physical block/cylinder locations is generally not possible on modern drives. But as a rough approximation, algorithms can assume that increasing LBAs mean increasing physical addresses, and LBAs close together equate to physical block proximity.

### 11.2.1 FCFS Scheduling

The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm (or FIFO). This algorithm is intrinsically fair, but it generally does not provide the fastest service. Consider, for example, a disk queue with requests for I/O to blocks on cylinders

98, 183, 37, 122, 14, 124, 65, 67,

in that order. If the disk head is initially at cylinder 53, it will first move from 53 to 98, then to 183, 37, 122, 14, 124, 65, and finally to 67, for a total head movement of 640 cylinders. This schedule is diagrammed in Figure 11.6.

The wild swing from 122 to 14 and then back to 124 illustrates the problem with this schedule. If the requests for cylinders 37 and 14 could be serviced together, before or after the requests for 122 and 124, the total head movement could be decreased substantially, and performance could be thereby improved.

### 11.2.2 SCAN Scheduling

In the **SCAN algorithm**, the disk arm starts at one end of the disk and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk. The SCAN algorithm is sometimes called the **elevator algorithm**, since the disk arm behaves just like an elevator in a building, first servicing all the requests going up and then reversing to service requests the other way.

Let's return to our example to illustrate. Before applying SCAN to schedule the requests on cylinders 98, 183, 37, 122, 14, 124, 65, and 67, we need to know

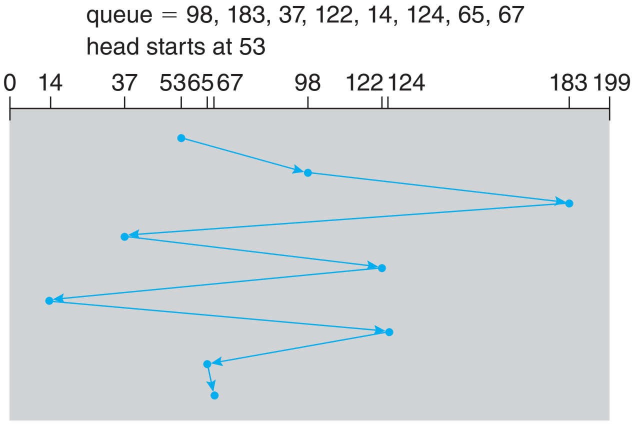

queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
0
14
37
53
65
67
98
122
124
183
199

**Figure 11.6** FCFS disk scheduling.

the direction of head movement in addition to the head's current position. Assuming that the disk arm is moving toward 0 and that the initial head position is again 53, the head will next service 37 and then 14. At cylinder 0, the arm will reverse and will move toward the other end of the disk, servicing the requests at 65, 67, 98, 122, 124, and 183 (Figure 11.7). If a request arrives in the queue just in front of the head, it will be serviced almost immediately; a request arriving just behind the head will have to wait until the arm moves to the end of the disk, reverses direction, and comes back.

Assuming a uniform distribution of requests for cylinders, consider the density of requests when the head reaches one end and reverses direction. At this point, relatively few requests are immediately in front of the head, since these cylinders have recently been serviced. The heaviest density of requests is at the other end of the disk. These requests have also waited the longest, so why not go there first? That is the idea of the next algorithm.

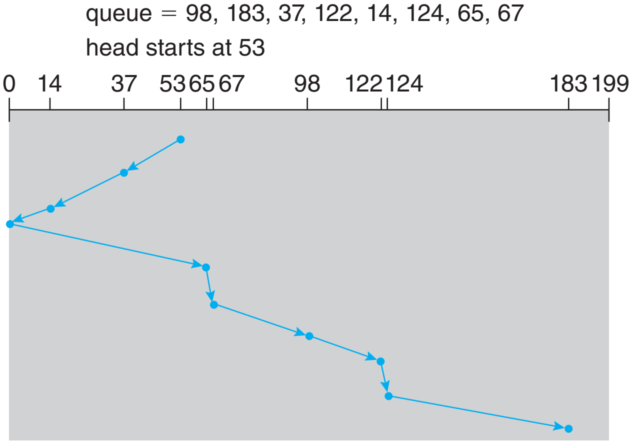

queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
0
14
37
53
65
67
98
122
124
183
199

**Figure 11.7** SCAN disk scheduling.

### 11.2.3 C-SCAN Scheduling

**Circular SCAN (C-SCAN) scheduling** is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk without servicing any requests on the return trip.

Let's return to our example to illustrate. Before applying C-SCAN to schedule the requests on cylinders 98, 183, 37, 122, 14, 124, 65, and 67, we need to know the direction of head movement in which the requests are scheduled. Assuming that the requests are scheduled when the disk arm is moving from 0 to 199 and that the initial head position is again 53, the request will be served as depicted in Figure 11.8. The C-SCAN scheduling algorithm essentially treats the cylinders as a circular list that wraps around from the final cylinder to the first one.

### 11.2.4 Selection of a Disk-Scheduling Algorithm

There are many disk-scheduling algorithms not included in this coverage, because they are rarely used. But how do operating system designers decide which to implement, and deployers chose the best to use? For any particular list of requests, we can define an optimal order of retrieval, but the computation needed to find an optimal schedule may not justify the savings over SCAN. With any scheduling algorithm, however, performance depends heavily on the number and types of requests. For instance, suppose that the queue usually has just one outstanding request. Then, all scheduling algorithms behave the same, because they have only one choice of where to move the disk head: they all behave like FCFS scheduling.

SCAN and C-SCAN perform better for systems that place a heavy load on the disk, because they are less likely to cause a starvation problem. There can still be starvation though, which drove Linux to create the **deadline** scheduler. This scheduler maintains separate read and write queues, and gives reads priority because processes are more likely to block on read than write. The queues are

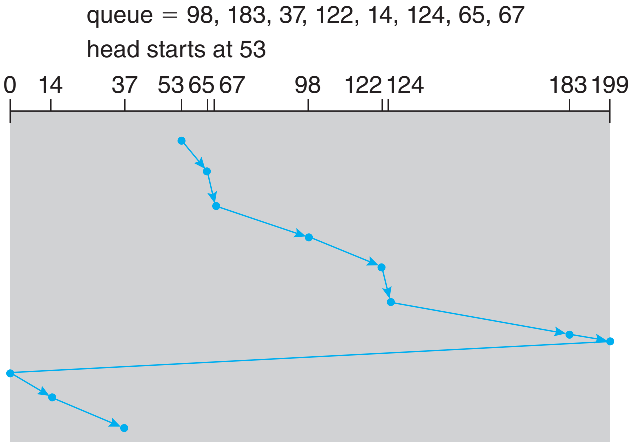


Figure 11.8 C-SCAN disk scheduling.

sorted in LBA order, essentially implementing C-SCAN. All I/O requests are sent in a batch in this LBA order. Deadline keeps four queues: two read and two write, one sorted by LBA and the other by FCFS. It checks after each batch to see if there are requests in the FCFS queues older than a configured age (by default, 500 ms). If so, the LBA queue (read or write) containing that request is selected for the next batch of I/O.

The deadline I/O scheduler is the default in the Linux RedHat 7 distribution, but **RHEL** 7 also includes two others. NOOP is preferred for CPU-bound systems using fast storage such as NVM devices, and the **Completely Fair Queueing scheduler** (**CFQ**) is the default for SATA drives. CFQ maintains three queues (with insertion sort to keep them sorted in LBA order): real time, best effort (the default), and idle. Each has exclusive priority over the others, in that order, with starvation possible. It uses historical data, anticipating if a process will likely issue more I/O requests soon. If it so determines, it idles waiting for the new I/O, ignoring other queued requests. This is to minimize seek time, assuming locality of reference of storage I/O requests, per process. Details of these schedulers can be found in https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Performance_Tuning_Guide/index.html.

## 11.3 NVM Scheduling

The disk-scheduling algorithms just discussed apply to mechanical platter-based storage like HDDs. They focus primarily on minimizing the amount of disk head movement. NVM devices do not contain moving disk heads and commonly use a simple FCFS policy. For example, the Linux **NOOP** scheduler uses an FCFS policy but modifies it to merge adjacent requests. The observed behavior of NVM devices indicates that the time required to service reads is uniform but that, because of the properties of flash memory, write service time is not uniform. Some SSD schedulers have exploited this property and merge only adjacent write requests, servicing all read requests in FCFS order.

As we have seen, I/O can occur sequentially or randomly. Sequential access is optimal for mechanical devices like HDD and tape because the data to be read or written is near the read/write head. Random-access I/O, which is measured in **input/output operations per second** (**IOPS**), causes HDD disk head movement. Naturally, random access I/O is much faster on NVM. An HDD can produce hundreds of IOPS, while an SSD can produce hundreds of thousands of IOPS.

NVM devices offer much less of an advantage for raw sequential throughput, where HDD head seeks are minimized and reading and writing of data to the media are emphasized. In those cases, for reads, performance for the two types of devices can range from equivalent to an order of magnitude advantage for NVM devices. Writing to NVM is slower than reading, decreasing the advantage. Furthermore, while write performance for HDDs is consistent throughout the life of the device, write performance for NVM devices varies depending on how full the device is (recall the need for garbage collection and over-provisioning) and how “worn” it is. An NVM device near its end of life due to many erase cycles generally has much worse performance than a new device.

One way to improve the lifespan and performance of NVM devices over time is to have the file system inform the device when files are deleted, so that the device can erase the blocks those files were stored on. This approach is discussed further in Section 14.5.6.

Let's look more closely at the impact of garbage collection on performance. Consider an NVM device under random read and write load. Assume that all blocks have been written to, but there is free space available. Garbage collection must occur to reclaim space taken by invalid data. That means that a write might cause a read of one or more pages, a write of the good data in those pages to overprovisioning space, an erase of the all-invalid-data block, and the placement of that block into overprovisioning space. In summary, one write request eventually causes a page write (the data), one or more page reads (by garbage collection), and one or more page writes (of good data from the garbage-collected blocks). The creation of I/O requests not by applications but by the NVM device doing garbage collection and space management is called **write amplificatio** and can greatly impact the write performance of the device. In the worst case, several extra I/Os are triggered with each write request.

## 11.4 Error Detection and Correction

Error detection and correction are fundamental to many areas of computing, including memory, networking, and storage. **Error detection** determines if a problem has occurred — for example a bit in DRAM spontaneously changed from a 0 to a 1, the contents of a network packet changed during transmission, or a block of data changed between when it was written and when it was read. By detecting the issue, the system can halt an operation before the error is propagated, report the error to the user or administrator, or warn of a device that might be starting to fail or has already failed.

Memory systems have long detected certain errors by using parity bits. In this scenario, each byte in a memory system has a parity bit associated with it that records whether the number of bits in the byte set to 1 is even (parity = 0) or odd (parity = 1). If one of the bits in the byte is damaged (either a 1 becomes a 0, or a 0 becomes a 1), the parity of the byte changes and thus does not match the stored parity. Similarly, if the stored parity bit is damaged, it does not match the computed parity. Thus, all single-bit errors are detected by the memory system. A double-bit-error might go undetected, however. Note that parity is easily calculated by performing an XOR (for "eXclusive OR") of the bits. Also note that for every byte of memory, we now need an extra bit of memory to store the parity.

Parity is one form of **checksums**, which use modular arithmetic to compute, store, and compare values on fixed-length words. Another error-detection method, common in networking, is a **cyclic redundancy check** (**CRCs**), which uses a hash function to detect multiple-bit errors (see http://www.mathpages.com/home/kmath458/kmath458.htm).

An **error-correction code** (**ECC**) not only detects the problem, but also corrects it. The correction is done by using algorithms and extra amounts of storage. The codes vary based on how much extra storage they need and how many errors they can correct. For example, disks drives use per-sector ECC and flash drives per-page ECC. When the controller writes a sector/page of data during normal I/O, the ECC is written with a value calculated from all the bytes in the data being written. When the sector/page is read, the ECC is recalculated and compared with the stored value. If the stored and calculated numbers are different, this mismatch indicates that the data have become corrupted and that the storage media may be bad (Section 11.5.3). The ECC is error correcting because it contains enough information, if only a few bits of data have been corrupted, to enable the controller to identify which bits have changed and calculate what their correct values should be. It then reports a recoverable **soft error**. If too many changes occur, and the ECC cannot correct the error, a non-correctable **hard error** is signaled. The controller automatically does the ECC processing whenever a sector or page is read or written.

Error detection and correction are frequently differentiators between consumer products and enterprise products. ECC is used in some systems for DRAM error correction and data path protection, for example.

## 11.5 Storage Device Management

The operating system is responsible for several other aspects of storage device management, too. Here, we discuss drive initialization, booting from a drive, and bad-block recovery.

### 11.5.1 Drive Formatting, Partitions, and Volumes

A new storage device is a blank slate: it is just a platter of a magnetic recording material or a set of uninitialized semiconductor storage cells. Before a storage device can store data, it must be divided into sectors that the controller can read and write. NVM pages must be initialized and the FTL created. This process is called **low-level formatting**, or **physical formatting**. Low-level formatting fills the device with a special data structure for each storage location. The data structure for a sector or page typically consists of a header, a data area, and a trailer. The header and trailer contain information used by the controller, such as a sector/page number and an error detection or correction code.

Most drives are low-level-formatted at the factory as a part of the manufacturing process. This formatting enables the manufacturer to test the device and to initialize the mapping from logical block numbers to defect-free sectors or pages on the media. It is usually possible to choose among a few sector sizes, such as 512 bytes and 4KB. Formatting a disk with a larger sector size means that fewer sectors can fit on each track, but it also means that fewer headers and trailers are written on each track and more space is available for user data. Some operating systems can handle only one specific sector size.

Before it can use a drive to hold files, the operating system still needs to record its own data structures on the device. It does so in three steps.

The first step is to **partition** the device into one or more groups of blocks or pages. The operating system can treat each partition as though it were a separate device. For instance, one partition can hold a file system containing a copy of the operating system's executable code, another the swap space, and another a file system containing the user files. Some operating systems and file systems perform the partitioning automatically when an entire device is to be managed by the file system. The partition information is written in a fixed format at a fixed location on the storage device. In Linux, the `fdisk` command is used to manage partitions on storage devices. The device, when recognized by the operating system, has its partition information read, and the operating system then creates device entries for the partitions (in `/dev` in Linux). From there, a configuration file, such as `/etc/fstab`, tells the operating system to mount each partition containing a file system at a specified location and to use mount options such as read-only. **Mounting** a file system is making the file system available for use by the system and its users.

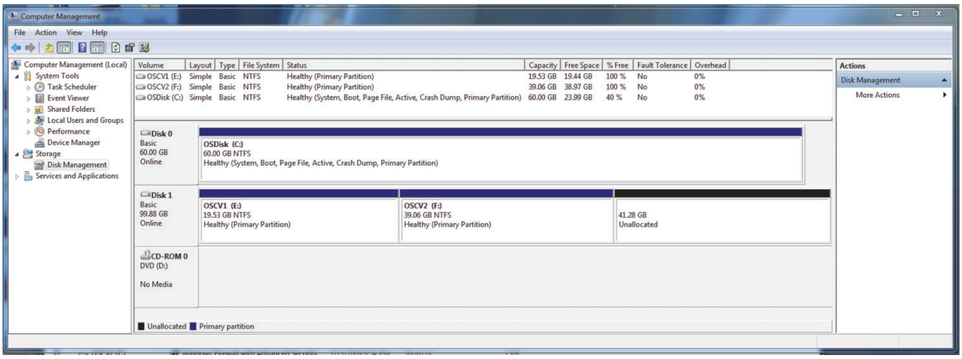
The second step is volume creation and management. Sometimes, this step is implicit, as when a file system is placed directly within a partition. That **volume** is then ready to be mounted and used. At other times, volume creation and management is explicit—for example when multiple partitions or devices will be used together as a RAID set (see Section 11.8) with one or more file systems spread across the devices. The Linux volume manager `lvm2` can provide these features, as can commercial third-party tools for Linux and other operating systems. ZFS provides both volume management and a file system integrated into one set of commands and features. (Note that "volume" can also mean any mountable file system, even a file containing a file system such as a CD image.)

The third step is **logical formatting**, or creation of a file system. In this step, the operating system stores the initial file-system data structures onto the device. These data structures may include maps of free and allocated space and an initial empty directory.

The partition information also indicates if a partition contains a bootable file system (containing the operating system). The partition labeled for boot is used to establish the root of the file system. Once it is mounted, device links for all other devices and their partitions can be created. Generally, a computer's "file system" consists of all mounted volumes. On Windows, these are separately named via a letter (C:, D:, E:). On other systems, such as Linux, at boot time the boot file system is mounted, and other file systems can be mounted within that tree structure (as discussed in Section 13.3). On Windows, the file system interface makes it clear when a given device is being used, while in Linux a single file access might traverse many devices before the requested file in the requested file system (within a volume) is accessed. Figure 11.9 shows the Windows 7 `Disk Management` tool displaying three volumes (C:, E:, and F:). Note that E: and F: are each in a partition of the "Disk 1" device and that there is unallocated space on that device for more partitions (possibly containing file systems).

To increase efficiency, most file systems group blocks together into larger chunks, frequently called **clusters**. Device I/O is done via blocks, but file system I/O is done via clusters, effectively assuring that I/O has more sequential-access and fewer random-access characteristics. File systems try to group file contents near its metadata as well, reducing HDD head seeks when operating on a file, for example.

Some operating systems give special programs the ability to use a partition as a large sequential array of logical blocks, without any file-system data structures. This array is sometimes called the **raw disk**, and I/O to this array is termed ***raw I/O***. It can be used for swap space (see Section 11.6.2), for example, and some database systems prefer raw I/O because it enables them to control



| Volume | Layout | Type | File System | Status | Capacity | Free Space | % Free | Fault Tolerance | Overhead |
|---|---|---|---|---|---|---|---|---|---|
| OSCV1 (E:) | Simple | Basic | NTFS | Healthy (Primary Partition) | 19.53 GB | 19.44 GB | 100 % | No | 0% |
| OSCV2 (F:) | Simple | Basic | NTFS | Healthy (Primary Partition) | 39.06 GB | 38.97 GB | 100 % | No | 0% |
| OSDisk (C:) | Simple | Basic | NTFS | Healthy (System, Boot, Page File, Active, Crash Dump, Primary Partition) | 60.00 GB | 23.99 GB | 40 % | No | 0% |

Disk 0
Basic
60.00 GB
Online

OSDisk (C:)
60.00 GB NTFS
Healthy (System, Boot, Page File, Active, Crash Dump, Primary Partition)

Disk 1
Basic
99.88 GB
Online

OSCV1 (E:)
19.53 GB NTFS
Healthy (Primary Partition)

OSCV2 (F:)
39.06 GB NTFS
Healthy (Primary Partition)

41.28 GB
Unallocated

CD-ROM 0
DVD (D:)

No Media

**Figure 11.9** Windows 7 `Disk Management` tool showing devices, partitions, volumes, and file systems.

the exact location where each database record is stored. Raw I/O bypasses all the file-system services, such as the buffer cache, file locking, prefetching, space allocation, file names, and directories. We can make certain applications more efficient by allowing them to implement their own special-purpose storage services on a raw partition, but most applications use a provided file system rather than managing data themselves. Note that Linux generally does not support raw I/O but can achieve similar access by using the DIRECT flag to the `open()` system call.

### 11.5.2 Boot Block

For a computer to start running—for instance, when it is powered up or rebooted—it must have an initial program to run. This initial **bootstrap** loader tends to be simple. For most computers, the bootstrap is stored in NVM flash memory firmware on the system motherboard and mapped to a known memory location. It can be updated by product manufacturers as needed, but also can be written to by viruses, infecting the system. It initializes all aspects of the system, from CPU registers to device controllers and the contents of main memory.

This tiny bootstrap loader program is also smart enough to bring in a full bootstrap program from secondary storage. The full bootstrap program is stored in the “boot blocks” at a fixed location on the device. The default Linux bootstrap loader is grub2 (https://www.gnu.org/software/grub/manual/grub.html/). A device that has a boot partition is called a **boot disk** or **system disk**.

The code in the bootstrap NVM instructs the storage controller to read the boot blocks into memory (no device drivers are loaded at this point) and then starts executing that code. The full bootstrap program is more sophisticated than the bootstrap loader: it is able to load the entire operating system from a non-fixed location on the device and to start the operating system running.

Let’s consider as an example the boot process in Windows. First, note that Windows allows a drive to be divided into partitions, and one partition—identified as the **boot partition**—contains the operating system and device drivers. The Windows system places its boot code in the first logical block on the hard disk or first page of the NVM device, which it terms the **master boot**

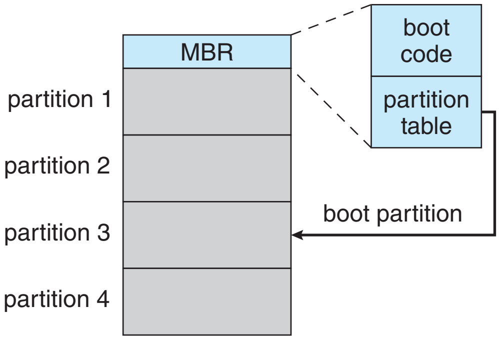


**Figure 11.10** Booting from a storage device in Windows.

**record**, or **MBR**. Booting begins by running code that is resident in the system's firmware. This code directs the system to read the boot code from the MBR, understanding just enough about the storage controller and storage device to load a sector from it. In addition to containing boot code, the MBR contains a table listing the partitions for the drive and a flag indicating which partition the system is to be booted from, as illustrated in Figure 11.10. Once the system identifies the boot partition, it reads the first sector/page from that partition (called the **boot sector**), which directs it to the kernel. It then continues with the remainder of the boot process, which includes loading the various subsystems and system services.

### 11.5.3 Bad Blocks

Because disks have moving parts and small tolerances (recall that the disk head flies just above the disk surface), they are prone to failure. Sometimes the failure is complete; in this case, the disk needs to be replaced and its contents restored from backup media to the new disk. More frequently, one or more sectors become defective. Most disks even come from the factory with **bad blocks**. Depending on the disk and controller in use, these blocks are handled in a variety of ways.

On older disks, such as some disks with IDE controllers, bad blocks are handled manually. One strategy is to scan the disk to find bad blocks while the disk is being formatted. Any bad blocks that are discovered are flagged as unusable so that the file system does not allocate them. If blocks go bad during normal operation, a special program (such as the Linux `badblocks` command) must be run manually to search for the bad blocks and to lock them away. Data that resided on the bad blocks usually are lost.

More sophisticated disks are smarter about bad-block recovery. The controller maintains a list of bad blocks on the disk. The list is initialized during the low-level formatting at the factory and is updated over the life of the disk. Low-level formatting also sets aside spare sectors not visible to the operating system. The controller can be told to replace each bad sector logically with one of the spare sectors. This scheme is known as **sector sparing** or **forwarding**.

A typical bad-sector transaction might be as follows:

- The operating system tries to read logical block 87.

- The controller calculates the ECC and finds that the sector is bad. It reports this finding to the operating system as an I/O error.
- The device controller replaces the bad sector with a spare.
- After that, whenever the system requests logical block 87, the request is translated into the replacement sector's address by the controller.

Note that such a redirection by the controller could invalidate any optimization by the operating system's disk-scheduling algorithm! For this reason, most disks are formatted to provide a few spare sectors in each cylinder and a spare cylinder as well. When a bad block is remapped, the controller uses a spare sector from the same cylinder, if possible.

As an alternative to sector sparing, some controllers can be instructed to replace a bad block by **sector slipping**. Here is an example: Suppose that logical block 17 becomes defective and the first available spare follows sector 202. Sector slipping then remaps all the sectors from 17 to 202, moving them all down one spot. That is, sector 202 is copied into the spare, then sector 201 into 202, then 200 into 201, and so on, until sector 18 is copied into sector 19. Slipping the sectors in this way frees up the space of sector 18 so that sector 17 can be mapped to it.

Recoverable soft errors may trigger a device activity in which a copy of the block data is made and the block is spared or slipped. An unrecoverable **hard error**, however, results in lost data. Whatever file was using that block must be repaired (for instance, by restoration from a backup tape), and that requires manual intervention.

NVM devices also have bits, bytes, and even pages that either are nonfunctional at manufacturing time or go bad over time. Management of those faulty areas is simpler than for HDDs because there is no seek time performance loss to be avoided. Either multiple pages can be set aside and used as replacement locations, or space from the over-provisioning area can be used (decreasing the usable capacity of the over-provisioning area). Either way, the controller maintains a table of bad pages and never sets those pages as available to write to, so they are never accessed.

## 11.6 Swap-Space Management

Swapping was first presented in Section 9.5, where we discussed moving entire processes between secondary storage and main memory. Swapping in that setting occurs when the amount of physical memory reaches a critically low point and processes are moved from memory to swap space to free available memory. In practice, very few modern operating systems implement swapping in this fashion. Rather, systems now combine swapping with virtual memory techniques (Chapter 10) and swap pages, not necessarily entire processes. In fact, some systems now use the terms “swapping” and “paging” interchangeably, reflecting the merging of these two concepts.

**Swap-space management** is another low-level task of the operating system. Virtual memory uses secondary storage space as an extension of main memory. Since drive access is much slower than memory access, using swap space significantly decreases system performance. The main goal for the design and implementation of swap space is to provide the best throughput for the virtual memory system. In this section, we discuss how swap space is used, where swap space is located on storage devices, and how swap space is managed.

### 11.6.1 Swap-Space Use

Swap space is used in various ways by different operating systems, depending on the memory-management algorithms in use. For instance, systems that implement swapping may use swap space to hold an entire process image, including the code and data segments. Paging systems may simply store pages that have been pushed out of main memory. The amount of swap space needed on a system can therefore vary from a few megabytes of disk space to gigabytes, depending on the amount of physical memory, the amount of virtual memory it is backing, and the way in which the virtual memory is used.

Note that it may be safer to overestimate than to underestimate the amount of swap space required, because if a system runs out of swap space it may be forced to abort processes or may crash entirely. Overestimation wastes secondary storage space that could otherwise be used for files, but it does no other harm. Some systems recommend the amount to be set aside for swap space. Solaris, for example, suggests setting swap space equal to the amount by which virtual memory exceeds pageable physical memory. In the past, Linux has suggested setting swap space to double the amount of physical memory. Today, the paging algorithms have changed, and most Linux systems use considerably less swap space.

Some operating systems—including Linux—allow the use of multiple swap spaces, including both files and dedicated swap partitions. These swap spaces are usually placed on separate storage devices so that the load placed on the I/O system by paging and swapping can be spread over the system's I/O bandwidth.

### 11.6.2 Swap-Space Location

A swap space can reside in one of two places: it can be carved out of the normal file system, or it can be in a separate partition. If the swap space is simply a large file within the file system, normal file-system routines can be used to create it, name it, and allocate its space.

Alternatively, swap space can be created in a separate **raw partition**. No file system or directory structure is placed in this space. Rather, a separate swap-space storage manager is used to allocate and deallocate the blocks from the raw partition. This manager uses algorithms optimized for speed rather than for storage efficiency, because swap space is accessed much more frequently than file systems, when it is used (recall that swap space is used for swapping and paging). Internal fragmentation may increase, but this tradeoff is acceptable because the life of data in the swap space generally is much shorter than that of files in the file system. Since swap space is reinitialized at boot time, any fragmentation is short-lived. The raw-partition approach creates a fixed amount of swap space during disk partitioning. Adding more swap space requires either repartitioning the device (which involves moving the other file-system partitions or destroying them and restoring them from backup) or adding another swap space elsewhere.

Some operating systems are flexible and can swap both in raw partitions and in file-system space. Linux is an example: the policy and implementation are separate, allowing the machine's administrator to decide which type of swapping to use. The trade-off is between the convenience of allocation and management in the file system and the performance of swapping in raw partitions.

### 11.6.3 Swap-Space Management: An Example

We can illustrate how swap space is used by following the evolution of swapping and paging in various UNIX systems. The traditional UNIX kernel started with an implementation of swapping that copied entire processes between contiguous disk regions and memory. UNIX later evolved to a combination of swapping and paging as paging hardware became available.

In Solaris 1 (SunOS), the designers changed standard UNIX methods to improve efficiency and reflect technological developments. When a process executes, text-segment pages containing code are brought in from the file system, accessed in main memory, and thrown away if selected for pageout. It is more efficient to reread a page from the file system than to write it to swap space and then reread it from there. Swap space is only used as a backing store for pages of **anonymous** memory (memory not backed by any file), which includes memory allocated for the stack, heap, and uninitialized data of a process.

More changes were made in later versions of Solaris. The biggest change is that Solaris now allocates swap space only when a page is forced out of physical memory, rather than when the virtual memory page is first created. This scheme gives better performance on modern computers, which have more physical memory than older systems and tend to page less.

Linux is similar to Solaris in that swap space is now used only for anonymous memory. Linux allows one or more swap areas to be established. A swap area may be in either a swap file on a regular file system or a dedicated swap partition. Each swap area consists of a series of 4-KB **page slots**, which are used to hold swapped pages. Associated with each swap area is a **swap map**—an array of integer counters, each corresponding to a page slot in the swap area. If the value of a counter is 0, the corresponding page slot is available. Values greater than 0 indicate that the page slot is occupied by a swapped page. The value of the counter indicates the number of mappings to the swapped page. For example, a value of 3 indicates that the swapped page is mapped to three different processes (which can occur if the swapped page is storing a region of memory shared by three processes). The data structures for swapping on Linux systems are shown in Figure 11.11.

## 11.7 Storage Attachment

Computers access secondary storage in three ways: via host-attached storage, network-attached storage, and cloud storage.

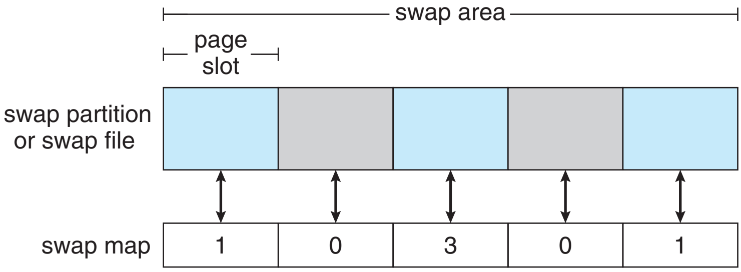


**Figure 11.11** The data structures for swapping on Linux systems.

### 11.7.1 Host-Attached Storage

**Host-attached storage** is storage accessed through local I/O ports. These ports use several technologies, the most common being SATA, as mentioned earlier. A typical system has one or a few SATA ports.

To allow a system to gain access to more storage, either an individual storage device, a device in a chassis, or multiple drives in a chassis can be connected via USB FireWire or Thunderbolt ports and cables.

High-end workstations and servers generally need more storage or need to share storage, so use more sophisticated I/O architectures, such as **fibr channel** (**FC**), a high-speed serial architecture that can operate over optical fiber or over a four-conductor copper cable. Because of the large address space and the switched nature of the communication, multiple hosts and storage devices can attach to the fabric, allowing great flexibility in I/O communication.

A wide variety of storage devices are suitable for use as host-attached storage. Among these are HDDs; NVM devices; CD, DVD, Blu-ray, and tape drives; and storage-area networks (**SANs**) (discussed in Section 11.7.4). The I/O commands that initiate data transfers to a host-attached storage device are reads and writes of logical data blocks directed to specifically identified storage units (such as bus ID or target logical unit).

### 11.7.2 Network-Attached Storage

**Network-attached storage** (**NAS**) (Figure 11.12) provides access to storage across a network. An NAS device can be either a special-purpose storage system or a general computer system that provides its storage to other hosts across the network. Clients access network-attached storage via a remote-procedure-call interface such as NFS for UNIX and Linux systems or CIFS for Windows machines. The remote procedure calls (RPCs) are carried via TCP or UDP over an IP network—usually the same local-area network (LAN) that carries all data traffic to the clients. The network-attached storage unit is usually implemented as a storage array with software that implements the RPC interface.

CIFS and NFS provide various locking features, allowing the sharing of files between hosts accessing a NAS with those protocols. For example, a user logged in to multiple NAS clients can access her home directory from all of those clients, simultaneously.

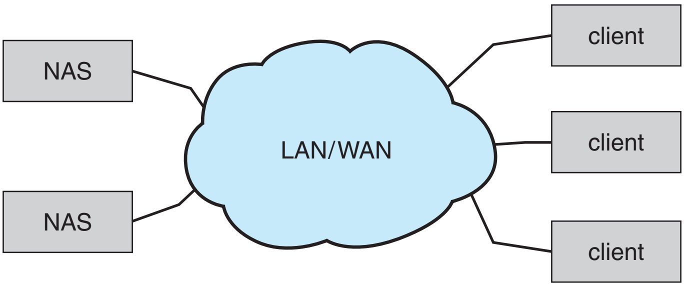


**Figure 11.12** Network-attached storage.

Network-attached storage provides a convenient way for all the computers on a LAN to share a pool of storage with the same ease of naming and access enjoyed with local host-attached storage. However, it tends to be less efficient and have lower performance than some direct-attached storage options.

**iSCSI** is the latest network-attached storage protocol. In essence, it uses the IP network protocol to carry the SCSI protocol. Thus, networks—rather than SCSI cables—can be used as the interconnects between hosts and their storage. As a result, hosts can treat their storage as if it were directly attached, even if the storage is distant from the host. Whereas NFS and CIFS present a file system and send parts of files across the network, iSCSI sends logical blocks across the network and leaves it to the client to use the blocks directly or create a file system with them.

### 11.7.3 Cloud Storage

Section 1.10.5 discussed cloud computing. One offering from cloud providers is **cloud storage**. Similar to network-attached storage, cloud storage provides access to storage across a network. Unlike NAS, the storage is accessed over the Internet or another WAN to a remote data center that provides storage for a fee (or even for free).

Another difference between NAS and cloud storage is how the storage is accessed and presented to users. NAS is accessed as just another file system if the CIFS or NFS protocols are used, or as a raw block device if the iSCSI protocol is used. Most operating systems have these protocols integrated and present NAS storage in the same way as other storage. In contrast, cloud storage is API based, and programs use the APIs to access the storage. Amazon S3 is a leading cloud storage offering. Dropbox is an example of a company that provides apps to connect to the cloud storage that it provides. Other examples include Microsoft OneDrive and Apple iCloud.

One reason that APIs are used instead of existing protocols is the latency and failure scenarios of a WAN. NAS protocols were designed for use in LANs, which have lower latency than WANs and are much less likely to lose connectivity between the storage user and the storage device. If a LAN connection fails, a system using NFS or CIFS might hang until it recovers. With cloud storage, failures like that are more likely, so an application simply pauses access until connectivity is restored.

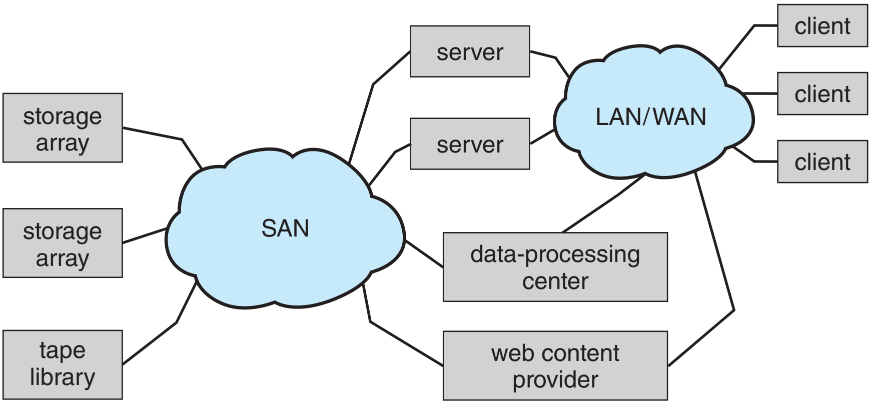


**Figure 11.13** Storage-area network.

### 11.7.4 Storage-Area Networks and Storage Arrays

One drawback of network-attached storage systems is that the storage I/O operations consume bandwidth on the data network, thereby increasing the latency of network communication. This problem can be particularly acute in large client–server installations—the communication between servers and clients competes for bandwidth with the communication among servers and storage devices.

A **storage-area network** (**SAN**) is a private network (using storage protocols rather than networking protocols) connecting servers and storage units, as shown in Figure 11.13. The power of a SAN lies in its flexibility. Multiple hosts and multiple storage arrays can attach to the same SAN, and storage can be dynamically allocated to hosts. The storage arrays can be RAID protected or unprotected drives (**Just a Bunch of Disks** (**JBOD**)). A SAN switch allows or prohibits access between the hosts and the storage. As one example, if a host is running low on disk space, the SAN can be configured to allocate more storage to that host. SANs make it possible for clusters of servers to share the same storage and for storage arrays to include multiple direct host connections. SANs typically have more ports—and cost more—than storage arrays. SAN connectivity is over short distances and typically has no routing, so a NAS can have many more connected hosts than a SAN.

A storage array is a purpose-built device (see Figure 11.14) that includes SAN ports, network ports, or both. It also contains drives to store data and a controller (or redundant set of controllers) to manage the storage and allow access to the storage across the networks. The controllers are composed of CPUs, memory, and software that implement the features of the array, which can include network protocols, user interfaces, RAID protection, snapshots, replication, compression, deduplication, and encryption. Some of those functions are discussed in Chapter 14.

Some storage arrays include SSDs. An array may contain only SSDs, resulting in maximum performance but smaller capacity, or may include a mix of SSDs and HDDs, with the array software (or the administrator) selecting the best medium for a given use or using the SSDs as a cache and HDDs as bulk storage.

**Figure 11.14** A storage array.

FC is the most common SAN interconnect, although the simplicity of iSCSI is increasing its use. Another SAN interconnect is **InfiniBan** (**IB**)—a special-purpose bus architecture that provides hardware and software support for high-speed interconnection networks for servers and storage units.

## 11.8 RAID Structure

Storage devices have continued to get smaller and cheaper, so it is now economically feasible to attach many drives to a computer system. Having a large number of drives in a system presents opportunities for improving the rate at which data can be read or written, if the drives are operated in parallel. Furthermore, this setup offers the potential for improving the reliability of data storage, because redundant information can be stored on multiple drives. Thus, failure of one drive does not lead to loss of data. A variety of disk-organization techniques, collectively called **redundant arrays of independent disks** (**RAIDs**), are commonly used to address the performance and reliability issues.

In the past, RAIDs composed of small, cheap disks were viewed as a cost-effective alternative to large, expensive disks. Today, RAIDs are used for their higher reliability and higher data-transfer rate rather than for economic reasons. Hence, the ***I*** in ***RAID***, which once stood for "inexpensive," now stands for "independent."

### 11.8.1 Improvement of Reliability via Redundancy

Let's first consider the reliability of a RAID of HDDs. The chance that some disk out of a set of *N* disks will fail is much greater than the chance that a specific single disk will fail. Suppose that the **mean time between failures** (**MTBF**) of a single disk is 100,000 hours. Then the MTBF of some disk in an array of 100

### *STRUCTURING RAID*

RAID storage can be structured in a variety of ways. For example, a system can have drives directly attached to its buses. In this case, the operating system or system software can implement RAID functionality. Alternatively, an intelligent host controller can control multiple attached devices and can implement RAID on those devices in hardware. Finally, a storage array can be used. A storage array, as just discussed, is a standalone unit with its own controller, cache, and drives. It is attached to the host via one or more standard controllers (for example, FC). This common setup allows an operating system or software without RAID functionality to have RAID-protected storage.

disks will be 100,000/100 = 1,000 hours, or 41.66 days, which is not long at all! If we store only one copy of the data, then each disk failure will result in loss of a significant amount of data—and such a high rate of data loss is unacceptable.

The solution to the problem of reliability is to introduce **redundancy**; we store extra information that is not normally needed but can be used in the event of disk failure to rebuild the lost information. Thus, even if a disk fails, data are not lost. RAID can be applied to NVM devices as well, although NVM devices have no moving parts and therefore are less likely to fail than HDDs.

The simplest (but most expensive) approach to introducing redundancy is to duplicate every drive. This technique is called **mirroring**. With mirroring, a logical disk consists of two physical drives, and every write is carried out on both drives. The result is called a **mirrored volume**. If one of the drives in the volume fails, the data can be read from the other. Data will be lost only if the second drive fails before the first failed drive is replaced.

The MTBF of a mirrored volume—where failure is the loss of data—depends on two factors. One is the MTBF of the individual drives. The other is the **mean time to repair**, which is the time it takes (on average) to replace a failed drive and to restore the data on it. Suppose that the failures of the two drives are independent; that is, the failure of one is not connected to the failure of the other. Then, if the MTBF of a single drive is 100,000 hours and the mean time to repair is 10 hours, the **mean time to data loss** of a mirrored drive system is $100,000^2/(2 * 10) = 500 * 10^6$ hours, or 57,000 years!

You should be aware that we cannot really assume that drive failures will be independent. Power failures and natural disasters, such as earthquakes, fires, and floods, may result in damage to both drives at the same time. Also, manufacturing defects in a batch of drives can cause correlated failures. As drives age, the probability of failure grows, increasing the chance that a second drive will fail while the first is being repaired. In spite of all these considerations, however, mirrored-drive systems offer much higher reliability than do single-drive systems.

Power failures are a particular source of concern, since they occur far more frequently than do natural disasters. Even with mirroring of drives, if writes are in progress to the same block in both drives, and power fails before both blocks are fully written, the two blocks can be in an inconsistent state. One solution to this problem is to write one copy first, then the next. Another is to add a solid-state nonvolatile cache to the RAID array. This write-back cache is protected from data loss during power failures, so the write can be considered complete at that point, assuming the cache has some kind of error protection and correction, such as ECC or mirroring.

### 11.8.2 Improvement in Performance via Parallelism

Now let's consider how parallel access to multiple drives improves performance. With mirroring, the rate at which read requests can be handled is doubled, since read requests can be sent to either drive (as long as both in a pair are functional, as is almost always the case). The transfer rate of each read is the same as in a single-drive system, but the number of reads per unit time has doubled.

With multiple drives, we can improve the transfer rate as well (or instead) by striping data across the drives. In its simplest form, **data striping** consists of splitting the bits of each byte across multiple drives; such striping is called **bit-level striping**. For example, if we have an array of eight drives, we write bit $i$ of each byte to drive $i$. The array of eight drives can be treated as a single drive with sectors that are eight times the normal size and, more important, have eight times the access rate. Every drive participates in every access (read or write); so the number of accesses that can be processed per second is about the same as on a single drive, but each access can read eight times as many data in the same time as on a single drive.

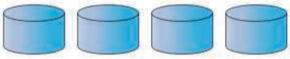
Bit-level striping can be generalized to include a number of drives that either is a multiple of 8 or divides 8. For example, if we use an array of four drives, bits $i$ and $4+i$ of each byte go to drive $i$. Further, striping need not occur at the bit level. In **block-level striping**, for instance, blocks of a file are striped across multiple drives; with $n$ drives, block $i$ of a file goes to drive $(i \bmod n)+1$. Other levels of striping, such as bytes of a sector or sectors of a block, also are possible. Block-level striping is the only commonly available striping.

Parallelism in a storage system, as achieved through striping, has two main goals:

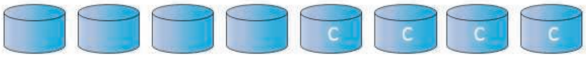
1. Increase the throughput of multiple small accesses (that is, page accesses) by load balancing.
2. Reduce the response time of large accesses.

### 11.8.3 RAID Levels

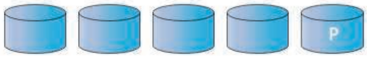
Mirroring provides high reliability, but it is expensive. Striping provides high data-transfer rates, but it does not improve reliability. Numerous schemes to provide redundancy at lower cost by using disk striping combined with "parity" bits (which we describe shortly) have been proposed. These schemes have different cost–performance trade-offs and are classified according to levels called **RAID levels**. We describe only the most common levels here; Figure 11.15 shows them pictorially (in the figure, $P$ indicates error-correcting bits and $C$ indicates a second copy of the data). In all cases depicted in the figure, four drives' worth of data are stored, and the extra drives are used to store redundant information for failure recovery.



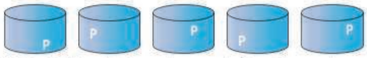
(a) RAID 0: non-redundant striping.



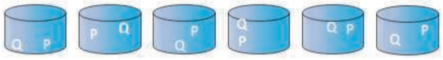
(b) RAID 1: mirrored disks.



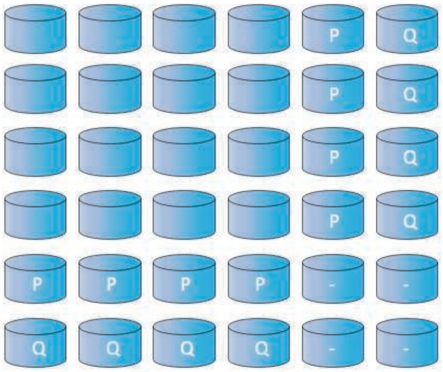
(c) RAID 4: block-interleaved parity.



(d) RAID 5: block-interleaved distributed parity.



(e) RAID 6: P + Q redundancy.



(f) Multidimensional RAID 6.

Figure 11.15 RAID levels.

- **RAID level 0**. RAID level 0 refers to drive arrays with striping at the level of blocks but without any redundancy (such as mirroring or parity bits), as shown in Figure 11.15(a).
- **RAID level 1**. RAID level 1 refers to drive mirroring. Figure 11.15(b) shows a mirrored organization.
- **RAID level 4**. RAID level 4 is also known as memory-style error-correcting-code (ECC) organization. ECC is also used in RAID 5 and 6.

  The idea of ECC can be used directly in storage arrays via striping of blocks across drives. For example, the first data block of a sequence of writes can be stored in drive 1, the second block in drive 2, and so on until the $N$th block is stored in drive $N$; the error-correction calculation result of those blocks is stored on drive $N + 1$. This scheme is shown in Figure 11.15(c), where the drive labeled $P$ stores the error-correction block. If one of the drives fails, the error-correction code recalculation detects that and prevents the data from being passed to the requesting process, throwing an error.

RAID 4 can actually correct errors, even though there is only one ECC block. It takes into account the fact that, unlike memory systems, drive controllers can detect whether a sector has been read correctly, so a single parity block can be used for error correction and detection. The idea is as follows: If one of the sectors is damaged, we know exactly which sector it is. We disregard the data in that sector and use the parity data to recalculate the bad data. For every bit in the block, we can determine if it is a 1 or a 0 by computing the parity of the corresponding bits from sectors in the other drives. If the parity of the remaining bits is equal to the stored parity, the missing bit is 0; otherwise, it is 1.

A block read accesses only one drive, allowing other requests to be processed by the other drives. The transfer rates for large reads are high, since all the disks can be read in parallel. Large writes also have high transfer rates, since the data and parity can be written in parallel.

Small independent writes cannot be performed in parallel. An operating-system write of data smaller than a block requires that the block be read, modified with the new data, and written back. The parity block has to be updated as well. This is known as the **read-modify-write cycle**. Thus, a single write requires four drive accesses: two to read the two old blocks and two to write the two new blocks.

WAFL (which we cover in Chapter 14) uses RAID level 4 because this RAID level allows drives to be added to a RAID set seamlessly. If the added drives are initialized with blocks containing only zeros, then the parity value does not change, and the RAID set is still correct.

RAID level 4 has two advantages over level 1 while providing equal data protection. First, the storage overhead is reduced because only one parity drive is needed for several regular drives, whereas one mirror drive is needed for every drive in level 1. Second, since reads and writes of a series of blocks are spread out over multiple drives with $N$-way striping of data, the transfer rate for reading or writing a set of blocks is $N$ times as fast as with level 1.

A performance problem with RAID 4—and with all parity-based RAID levels—is the expense of computing and writing the XOR parity. This overhead can result in slower writes than with non-parity RAID arrays. Modern general-purpose CPUs are very fast compared with drive I/O, however, so the performance hit can be minimal. Also, many RAID storage arrays or host bus-adapters include a hardware controller with dedicated parity hardware. This controller offloads the parity computation from the CPU to the array. The array has an NVRAM cache as well, to store the blocks while the parity is computed and to buffer the writes from the controller to the drives. Such buffering can avoid most read-modify-write cycles by gathering data to be written into a full stripe and writing to all drives in the stripe concurrently. This combination of hardware acceleration and buffering can make parity RAID almost as fast as non-parity RAID, frequently outperforming a non-caching non-parity RAID.

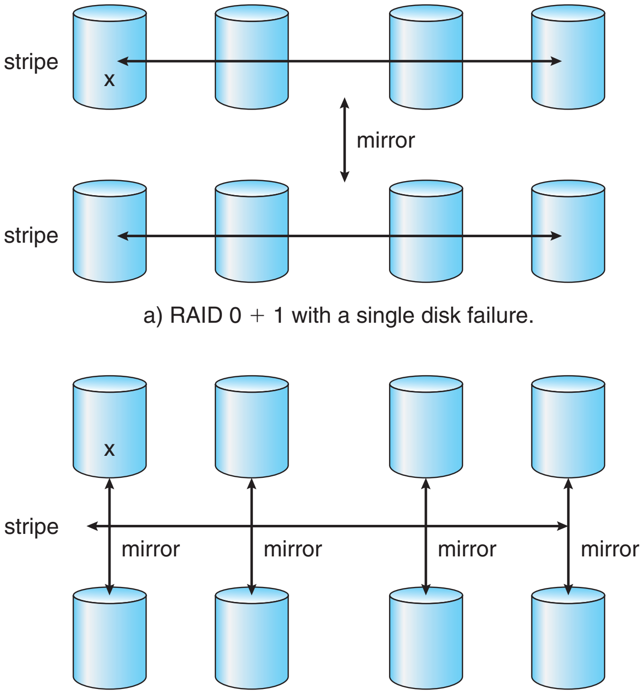
- **RAID level 5**. RAID level 5, or block-interleaved distributed parity, differs from level 4 in that it spreads data and parity among all $N+1$ drives, rather than storing data in *N* drives and parity in one drive. For each set of *N* blocks, one of the drives stores the parity and the others store data. For example, with an array of five drives, the parity for the *n*th block is stored in drive (*n* mod 5) + 1. The *n*th blocks of the other four drives store actual data for that block. This setup is shown in Figure 11.15(d), where the *P*s are distributed across all the drives. A parity block cannot store parity for blocks in the same drive, because a drive failure would result in loss of data as well as of parity, and hence the loss would not be recoverable. By spreading the parity across all the drives in the set, RAID 5 avoids potential overuse of a single parity drive, which can occur with RAID 4. RAID 5 is the most common parity RAID.

- **RAID level 6**. RAID level 6, also called the **P + Q redundancy scheme**, is much like RAID level 5 but stores extra redundant information to guard against multiple drive failures. XOR parity cannot be used on both parity blocks because they would be identical and would not provide more recovery information. Instead of parity, error-correcting codes such as **Galois field math** are used to calculate Q. In the scheme shown in Figure 11.15(e), 2 blocks of redundant data are stored for every 4 blocks of data—compared with 1 parity block in level 5—and the system can tolerate two drive failures.

- **Multidimensional RAID level 6**. Some sophisticated storage arrays amplify RAID level 6. Consider an array containing hundreds of drives. Putting those drives in a RAID level 6 stripe would result in many data drives and only two logical parity drives. Multidimensional RAID level 6 logically arranges drives into rows and columns (two or more dimensional arrays) and implements RAID level 6 both horizontally along the rows and vertically down the columns. The system can recover from any failure —or, indeed, multiple failures—by using parity blocks in any of these locations. This RAID level is shown in Figure 11.15(f). For simplicity, the figure shows the RAID parity on dedicated drives, but in reality the RAID blocks are scattered throughout the rows and columns.

- **RAID levels 0 + 1 and 1 + 0**. RAID level 0 + 1 refers to a combination of RAID levels 0 and 1. RAID 0 provides the performance, while RAID 1 provides the reliability. Generally, this level provides better performance than RAID 5. It is common in environments where both performance and reliability are important. Unfortunately, like RAID 1, it doubles the number of drives needed for storage, so it is also relatively expensive. In RAID 0 + 1, a set of drives are striped, and then the stripe is mirrored to another, equivalent stripe.

  Another RAID variation is RAID level 1 + 0, in which drives are mirrored in pairs and then the resulting mirrored pairs are striped. This scheme has some theoretical advantages over RAID 0 + 1. For example, if a single drive fails in RAID 0 + 1, an entire stripe is inaccessible, leaving only the other stripe. With a failure in RAID 1 + 0, a single drive is unavailable, but the drive that mirrors it is still available, as are all the rest of the drives (Figure 11.16).



a) RAID 0 + 1 with a single disk failure.

b) RAID 1 + 0 with a single disk failure.

**Figure 11.16** RAID 0 + 1 and 1 + 0 with a single disk failure.

Numerous variations have been proposed to the basic RAID schemes described here. As a result, some confusion may exist about the exact definitions of the different RAID levels.

The implementation of RAID is another area of variation. Consider the following layers at which RAID can be implemented.

- Volume-management software can implement RAID within the kernel or at the system software layer. In this case, the storage hardware can provide minimal features and still be part of a full RAID solution.
- RAID can be implemented in the host bus-adapter (HBA) hardware. Only the drives directly connected to the HBA can be part of a given RAID set. This solution is low in cost but not very flexible.
- RAID can be implemented in the hardware of the storage array. The storage array can create RAID sets of various levels and can even slice these sets into smaller volumes, which are then presented to the operating system. The operating system need only implement the file system on each of the volumes. Arrays can have multiple connections available or can be part of a SAN, allowing multiple hosts to take advantage of the array's features.
- RAID can be implemented in the SAN interconnect layer by drive virtualization devices. In this case, a device sits between the hosts and the storage. It

accepts commands from the servers and manages access to the storage. It could provide mirroring, for example, by writing each block to two separate storage devices.

Other features, such as snapshots and replication, can be implemented at each of these levels as well. A **snapshot** is a view of the file system before the last update took place. (Snapshots are covered more fully in Chapter 14.) **Replication** involves the automatic duplication of writes between separate sites for redundancy and disaster recovery. Replication can be synchronous or asynchronous. In synchronous replication, each block must be written locally and remotely before the write is considered complete, whereas in asynchronous replication, the writes are grouped together and written periodically. Asynchronous replication can result in data loss if the primary site fails, but it is faster and has no distance limitations. Increasingly, replication is also used within a data center or even within a host. As an alternative to RAID protection, replication protects against data loss and also increases read performance (by allowing reads from each of the replica copies). It does of course use more storage than most types of RAID.

The implementation of these features differs depending on the layer at which RAID is implemented. For example, if RAID is implemented in software, then each host may need to carry out and manage its own replication. If replication is implemented in the storage array or in the SAN interconnect, however, then whatever the host operating system or its features, the host's data can be replicated.

One other aspect of most RAID implementations is a hot spare drive or drives. A **hot spare** is not used for data but is configured to be used as a replacement in case of drive failure. For instance, a hot spare can be used to rebuild a mirrored pair should one of the drives in the pair fail. In this way, the RAID level can be reestablished automatically, without waiting for the failed drive to be replaced. Allocating more than one hot spare allows more than one failure to be repaired without human intervention.

### 11.8.4 Selecting a RAID Level

Given the many choices they have, how do system designers choose a RAID level? One consideration is rebuild performance. If a drive fails, the time needed to rebuild its data can be significant. This may be an important factor if a continuous supply of data is required, as it is in high-performance or interactive database systems. Furthermore, rebuild performance influences the mean time between failures.

Rebuild performance varies with the RAID level used. Rebuilding is easiest for RAID level 1, since data can be copied from another drive. For the other levels, we need to access all the other drives in the array to rebuild data in a failed drive. Rebuild times can be hours for RAID level 5 rebuilds of large drive sets.

RAID level 0 is used in high-performance applications where data loss is not critical. For example, in scientific computing where a data set is loaded and explored, RAID level 0 works well because any drive failures would just require a repair and reloading of the data from its source. RAID level 1 is popular for applications that require high reliability with fast recovery. RAID

### *THE InServ STORAGE ARRAY*

Innovation, in an effort to provide better, faster, and less expensive solutions, frequently blurs the lines that separated previous technologies. Consider the InServ storage array from HP 3Par. Unlike most other storage arrays, InServ does not require that a set of drives be configured at a specific RAID level. Rather, each drive is broken into 256-MB "chunklets." RAID is then applied at the chunklet level. A drive can thus participate in multiple and various RAID levels as its chunklets are used for multiple volumes.

InServ also provides snapshots similar to those created by the WAFL file system. The format of InServ snapshots can be read–write as well as read-only, allowing multiple hosts to mount copies of a given file system without needing their own copies of the entire file system. Any changes a host makes in its own copy are copy-on-write and so are not reflected in the other copies.

A further innovation is **utility storage**. Some file systems do not expand or shrink. On these systems, the original size is the only size, and any change requires copying data. An administrator can configure InServ to provide a host with a large amount of logical storage that initially occupies only a small amount of physical storage. As the host starts using the storage, unused drives are allocated to the host, up to the original logical level. The host thus can believe that it has a large fixed storage space, create its file systems there, and so on. Drives can be added to or removed from the file system by InServ without the file system's noticing the change. This feature can reduce the number of drives needed by hosts, or at least delay the purchase of drives until they are really needed.

0 + 1 and 1 + 0 are used where both performance and reliability are important —for example, for small databases. Due to RAID 1's high space overhead, RAID 5 is often preferred for storing moderate volumes of data. RAID 6 and multidimensional RAID 6 are the most common formats in storage arrays. They offer good performance and protection without large space overhead.

RAID system designers and administrators of storage have to make several other decisions as well. For example, how many drives should be in a given RAID set? How many bits should be protected by each parity bit? If more drives are in an array, data-transfer rates are higher, but the system is more expensive. If more bits are protected by a parity bit, the space overhead due to parity bits is lower, but the chance that a second drive will fail before the first failed drive is repaired is greater, and that will result in data loss.

### 11.8.5 Extensions

The concepts of RAID have been generalized to other storage devices, including arrays of tapes, and even to the broadcast of data over wireless systems. When applied to arrays of tapes, RAID structures are able to recover data even if one of the tapes in an array is damaged. When applied to broadcast of data, a block of data is split into short units and is broadcast along with a parity unit. If one of the units is not received for any reason, it can be reconstructed from the other units. Commonly, tape-drive robots containing multiple tape drives will stripe data across all the drives to increase throughput and decrease backup time.

### 11.8.6 Problems with RAID

Unfortunately, RAID does not always assure that data are available for the operating system and its users. A pointer to a file could be wrong, for example, or pointers within the file structure could be wrong. Incomplete writes (called "torn writes"), if not properly recovered, could result in corrupt data. Some other process could accidentally write over a file system's structures, too. RAID protects against physical media errors, but not other hardware and software errors. A failure of the hardware RAID controller, or a bug in the software RAID code, could result in total data loss. As large as is the landscape of software and hardware bugs, that is how numerous are the potential perils for data on a system.

The **Solaris ZFS** file system takes an innovative approach to solving these problems through the use of checksums. ZFS maintains internal checksums of all blocks, including data and metadata. These checksums are not kept with the block that is being checksummed. Rather, they are stored with the pointer to that block. (See Figure 11.17.) Consider an **inode**—a data structure for storing file system metadata—with pointers to its data. Within the inode is the checksum of each block of data. If there is a problem with the data, the checksum will be incorrect, and the file system will know about it. If the data are mirrored, and there is a block with a correct checksum and one with an incorrect checksum, ZFS will automatically update the bad block with the good one. Similarly, the directory entry that points to the inode has a checksum for the inode. Any problem in the inode is detected when the directory is accessed. This checksumming takes places throughout all ZFS structures, providing a much higher level of consistency, error detection, and error cor-

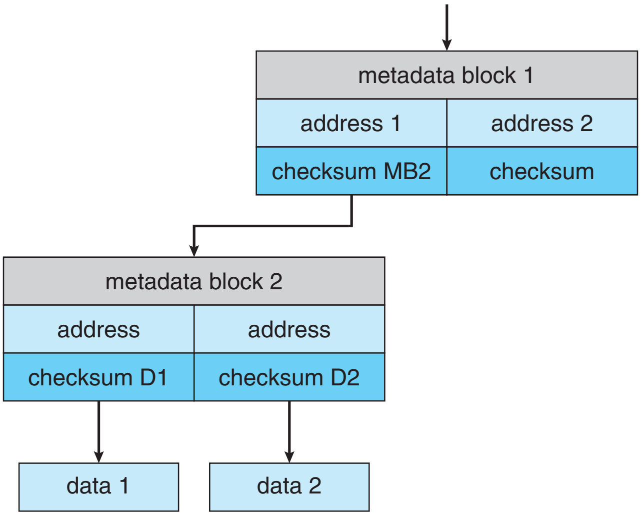


**Figure 11.17** ZFS checksums all metadata and data.

rection than is found in RAID drive sets or standard file systems. The extra overhead that is created by the checksum calculation and extra block read-modify-write cycles is not noticeable because the overall performance of ZFS is very fast. (A similar checksum feature is found in the Linux BTRFS file system. See https://btrfs.wiki.kernel.org/index.php/Btrfs_design.)

Another issue with most RAID implementations is lack of flexibility. Consider a storage array with twenty drives divided into four sets of five drives. Each set of five drives is a RAID level 5 set. As a result, there are four separate volumes, each holding a file system. But what if one file system is too large to fit on a five-drive RAID level 5 set? And what if another file system needs very little space? If such factors are known ahead of time, then the drives and volumes can be properly allocated. Very frequently, however, drive use and requirements change over time.

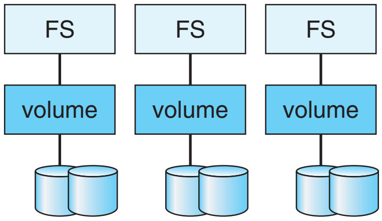
Even if the storage array allowed the entire set of twenty drives to be created as one large RAID set, other issues could arise. Several volumes of various sizes could be built on the set. But some volume managers do not allow us to change a volume's size. In that case, we would be left with the same issue described above—mismatched file-system sizes. Some volume managers allow size changes, but some file systems do not allow for file-system growth or shrinkage. The volumes could change sizes, but the file systems would need to be recreated to take advantage of those changes.

ZFS combines file-system management and volume management into a unit providing greater functionality than the traditional separation of those functions allows. Drives, or partitions of drives, are gathered together via RAID sets into **pools** of storage. A pool can hold one or more ZFS file systems. The entire pool's free space is available to all file systems within that pool. ZFS uses the memory model of `malloc()` and `free()` to allocate and release storage for each file system as blocks are used and freed within the file system. As a result, there are no artificial limits on storage use and no need to relocate file systems between volumes or resize volumes. ZFS provides quotas to limit the size of a file system and reservations to assure that a file system can grow by a specified amount, but those variables can be changed by the file-system owner at any time. Other systems like Linux have volume managers that allow the logical joining of multiple disks to create larger-than-disk volumes to hold large file systems. Figure 11.18(a) depicts traditional volumes and file systems, and Figure 11.18(b) shows the ZFS model.

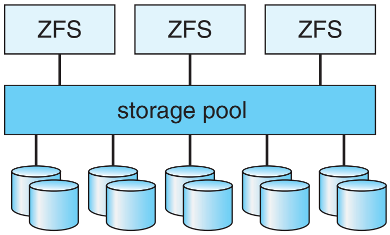
### 11.8.7 Object Storage

General-purpose computers typically use file systems to store content for users. Another approach to data storage is to start with a storage pool and place objects in that pool. This approach differs from file systems in that there is no way to navigate the pool and find those objects. Thus, rather than being user-oriented, object storage is computer-oriented, designed to be used by programs. A typical sequence is:

1. Create an object within the storage pool, and receive an object ID.
2. Access the object when needed via the object ID.
3. Delete the object via the object ID.



(a) Traditional volumes and file systems.



(b) ZFS and pooled storage.

**Figure 11.18** Traditional volumes and file systems compared with the ZFS model.

Object storage management software, such as the **Hadoop file system** (**HDFS**) and **Ceph**, determines where to store the objects and manages object protection. Typically, this occurs on commodity hardware rather than RAID arrays. For example, HDFS can store *N* copies of an object on *N* different computers. This approach can be lower in cost than storage arrays and can provide fast access to that object (at least on those *N* systems). All systems in a Hadoop cluster can access the object, but only systems that have a copy have fast access via the copy. Computations on the data occur on those systems, with results sent across the network, for example, only to the systems requesting them. Other systems need network connectivity to read and write to the object. Therefore, object storage is usually used for bulk storage, not high-speed random access. Object storage has the advantage of **horizontal scalability**. That is, whereas a storage array has a fixed maximum capacity, to add capacity to an object store, we simply add more computers with internal disks or attached external disks and add them to the pool. Object storage pools can be petabytes in size.

Another key feature of object storage is that each object is self-describing, including description of its contents. In fact, object storage is also known as **content-addressable storage**, because objects can be retrieved based on their contents. There is no set format for the contents, so what the system stores is **unstructured data**.

While object storage is not common on general-purpose computers, huge amounts of data are stored in object stores, including Google's Internet search contents, Dropbox contents, Spotify's songs, and Facebook photos. Cloud computing (such as Amazon AWS) generally uses object stores (in Amazon S3) to hold file systems as well as data objects for customer applications running on cloud computers.

For the history of object stores see http://www.theregister.co.uk/2016/07/15/the_history_boys_cas_and_object_storage_map.

## 11.9 Summary

- Hard disk drives and nonvolatile memory devices are the major secondary storage I/O units on most computers. Modern secondary storage is structured as large one-dimensional arrays of logical blocks.
- Drives of either type may be attached to a computer system in one of three ways: (1) through the local I/O ports on the host computer, (2) directly connected to motherboards, or (3) through a communications network or storage network connection.
- Requests for secondary storage I/O are generated by the file system and by the virtual memory system. Each request specifies the address on the device to be referenced in the form of a logical block number.
- Disk-scheduling algorithms can improve the effective bandwidth of HDDs, the average response time, and the variance in response time. Algorithms such as SCAN and C-SCAN are designed to make such improvements through strategies for disk-queue ordering. Performance of disk-scheduling algorithms can vary greatly on hard disks. In contrast, because solid-state disks have no moving parts, performance varies little among scheduling algorithms, and quite often a simple FCFS strategy is used.
- Data storage and transmission are complex and frequently result in errors. Error detection attempts to spot such problems to alert the system for corrective action and to avoid error propagation. Error correction can detect and repair problems, depending on the amount of correction data available and the amount of data that was corrupted.
- Storage devices are partitioned into one or more chunks of space. Each partition can hold a volume or be part of a multidevice volume. File systems are created in volumes.
- The operating system manages the storage device's blocks. New devices typically come pre-formatted. The device is partitioned, file systems are created, and boot blocks are allocated to store the system's bootstrap program if the device will contain an operating system. Finally, when a block or page is corrupted, the system must have a way to lock out that block or to replace it logically with a spare.
- An efficient swap space is a key to good performance in some systems. Some systems dedicate a raw partition to swap space, and others use a file within the file system instead. Still other systems allow the user or system administrator to make the decision by providing both options.
- Because of the amount of storage required on large systems, and because storage devices fail in various ways, secondary storage devices are frequently made redundant via RAID algorithms. These algorithms allow more than one drive to be used for a given operation and allow continued

operation and even automatic recovery in the face of a drive failure. RAID algorithms are organized into different levels; each level provides some combination of reliability and high transfer rates.

- Object storage is used for big data problems such as indexing the Internet and cloud photo storage. Objects are self-defining collections of data, addressed by object ID rather than file name. Typically it uses replication for data protection, computes based on the data on systems where a copy of the data exists, and is horizontally scalable for vast capacity and easy expansion.

## Practice Exercises

**11.1** Is disk scheduling, other than FCFS scheduling, useful in a single-user environment? Explain your answer.

**11.2** Explain why SSTF scheduling tends to favor middle cylinders over the innermost and outermost cylinders.

**11.3** Why is rotational latency usually not considered in disk scheduling? How would you modify SSTF, SCAN, and C-SCAN to include latency optimization?

**11.4** Why is it important to balance file-system I/O among the disks and controllers on a system in a multitasking environment?

**11.5** What are the tradeoffs involved in rereading code pages from the file system versus using swap space to store them?

**11.6** Is there any way to implement truly stable storage? Explain your answer.

**11.7** It is sometimes said that tape is a sequential-access medium, whereas a hard disk is a random-access medium. In fact, the suitability of a storage device for random access depends on the transfer size. The term ***streaming transfer rate*** denotes the rate for a data transfer that is underway, excluding the effect of access latency. In contrast, the ***effective transfer rate*** is the ratio of total bytes to total seconds, including overhead time such as access latency.

Suppose we have a computer with the following characteristics: the level-2 cache has an access latency of 8 nanoseconds and a streaming transfer rate of 800 megabytes per second, the main memory has an access latency of 60 nanoseconds and a streaming transfer rate of 80 megabytes per second, the hard disk has an access latency of 15 milliseconds and a streaming transfer rate of 5 megabytes per second, and a tape drive has an access latency of 60 seconds and a streaming transfer rate of 2 megabytes per second.

a. Random access causes the effective transfer rate of a device to decrease, because no data are transferred during the access time. For the disk described, what is the effective transfer rate if an average access is followed by a streaming transfer of (1) 512 bytes, (2) 8 kilobytes, (3) 1 megabyte, and (4) 16 megabytes?

b. The utilization of a device is the ratio of effective transfer rate to streaming transfer rate. Calculate the utilization of the disk drive for each of the four transfer sizes given in part a.

c. Suppose that a utilization of 25 percent (or higher) is considered acceptable. Using the performance figures given, compute the smallest transfer size for a disk that gives acceptable utilization.

d. Complete the following sentence: A disk is a random-access device for transfers larger than ______ bytes and is a sequential-access device for smaller transfers.

e. Compute the minimum transfer sizes that give acceptable utilization for cache, memory, and tape.

f. When is a tape a random-access device, and when is it a sequential-access device?

**11.8** Could a RAID level 1 organization achieve better performance for read requests than a RAID level 0 organization (with nonredundant striping of data)? If so, how?

**11.9** Give three reasons to use HDDs as secondary storage.

**11.10** Give three reasons to use NVM devices as secondary storage.

## Further Reading

[Services (2012)] provides an overview of data storage in a variety of modern computing environments. Discussions of redundant arrays of independent disks (RAIDs) are presented by [Patterson et al. (1988)]. [Kim et al. (2009)] discuss disk-scheduling algorithms for SSDs. Object-based storage is described by [Mesnier et al. (2003)].

[Russinovich et al. (2017)], [McDougall and Mauro (2007)], and [Love (2010)] discuss file-system details in Windows, Solaris, and Linux, respectively.

Storage devices are continuously evolving, with goals of increasing performance, increasing capacity, or both. For one direction in capacity improvement see http://www.tomsitpro.com/articles/shingled-magnetic-recoding-smr-101-basics,2-933.html).

RedHat (and other) Linux distributions have multiple, selectable disk scheduling algorithms. For details see https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Performance_Tuning_Guide/index.html.

Learn more about the default Linux bootstrap loader at https://www.gnu.org/software/grub/manual/grub.html/.

A relatively new file system, BTRFS, is detailed in https://btrfs.wiki.kernel.org/index.php/Btrfs_design.

For the history of object stores see http://www.theregister.co.uk/2016/07/15/the_history_boys_cas_and_object_storage_map.

# Bibliography

**[Kim et al. (2009)]** J. Kim, Y. Oh, E. Kim, J. C. D. Lee, and S. Noh, "Disk Schedulers for Solid State Drivers", *Proceedings of the seventh ACM international conference on Embedded software* (2009), pages 295–304.

**[Love (2010)]** R. Love, *Linux Kernel Development,* Third Edition, Developer's Library (2010).

**[McDougall and Mauro (2007)]** R. McDougall and J. Mauro, *Solaris Internals,* Second Edition, Prentice Hall (2007).

**[Mesnier et al. (2003)]** M. Mesnier, G. Ganger, and E. Ridel, "Object-based storage", *IEEE Communications Magazine*, Volume 41, Number 8 (2003), pages 84–99.

**[Patterson et al. (1988)]** D. A. Patterson, G. Gibson, and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", *Proceedings of the ACM SIGMOD International Conference on the Management of Data* (1988), pages 109–116.

**[Russinovich et al. (2017)]** M. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals–Part 1,* Seventh Edition, Microsoft Press (2017).

**[Services (2012)]** E. E. Services, *Information Storage and Management: Storing, Managing, and Protecting Digital Information in Classic, Virtualized, and Cloud Environments*, Wiley (2012).

## Chapter 11 Exercises

**11.11** None of the disk-scheduling disciplines, except FCFS, is truly fair (starvation may occur).

a. Explain why this assertion is true.

b. Describe a way to modify algorithms such as SCAN to ensure fairness.

c. Explain why fairness is an important goal in a multi-user systems.

d. Give three or more examples of circumstances in which it is important that the operating system be unfair in serving I/O requests.

**11.12** Explain why NVM devices often use an FCFS disk-scheduling algorithm.

**11.13** Suppose that a disk drive has 5,000 cylinders, numbered 0 to 4,999. The drive is currently serving a request at cylinder 2,150, and the previous request was at cylinder 1,805. The queue of pending requests, in FIFO order, is:

2,069; 1,212; 2,296; 2,800; 544; 1,618; 356; 1,523; 4,965; 3,681

Starting from the current head position, what is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests for each of the following disk-scheduling algorithms?

a. FCFS

b. SCAN

c. C-SCAN

**11.14** Elementary physics states that when an object is subjected to a constant acceleration $a$, the relationship between distance $d$ and time $t$ is given by $d = \frac{1}{2}at^2$. Suppose that, during a seek, the disk in Exercise 11.14 accelerates the disk arm at a constant rate for the first half of the seek, then decelerates the disk arm at the same rate for the second half of the seek. Assume that the disk can perform a seek to an adjacent cylinder in 1 millisecond and a full-stroke seek over all 5,000 cylinders in 18 milliseconds.

a. The distance of a seek is the number of cylinders over which the head moves. Explain why the seek time is proportional to the square root of the seek distance.

b. Write an equation for the seek time as a function of the seek distance. This equation should be of the form $t = x + y\sqrt{L}$, where $t$ is the time in milliseconds and $L$ is the seek distance in cylinders.

c. Calculate the total seek time for each of the schedules in Exercise 11.14. Determine which schedule is the fastest (has the smallest total seek time).

d. The **percentage speedup** is the time saved divided by the original time. What is the percentage speedup of the fastest schedule over FCFS?

**11.15** Suppose that the disk in Exercise 11.15 rotates at 7,200 RPM.

a. What is the average rotational latency of this disk drive?

b. What seek distance can be covered in the time that you found for part a?

**11.16** Compare and contrast HDDs and NVM devices. What are the best applications for each type?

**11.17** Describe some advantages and disadvantages of using NVM devices as a caching tier and as a disk-drive replacement compared with using only HDDs.

**11.18** Compare the performance of C-SCAN and SCAN scheduling, assuming a uniform distribution of requests. Consider the average response time (the time between the arrival of a request and the completion of that request's service), the variation in response time, and the effective bandwidth. How does performance depend on the relative sizes of seek time and rotational latency?

**11.19** Requests are not usually uniformly distributed. For example, we can expect a cylinder containing the file-system metadata to be accessed more frequently than a cylinder containing only files. Suppose you know that 50 percent of the requests are for a small, fixed number of cylinders.

a. Would any of the scheduling algorithms discussed in this chapter be particularly good for this case? Explain your answer.

b. Propose a disk-scheduling algorithm that gives even better performance by taking advantage of this "hot spot" on the disk.

**11.20** Consider a RAID level 5 organization comprising five disks, with the parity for sets of four blocks on four disks stored on the fifth disk. How many blocks are accessed in order to perform the following?

a. A write of one block of data

b. A write of seven continuous blocks of data

**11.21** Compare the throughput achieved by a RAID level 5 organization with that achieved by a RAID level 1 organization for the following:

a. Read operations on single blocks

b. Read operations on multiple contiguous blocks

**11.22** Compare the performance of write operations achieved by a RAID level 5 organization with that achieved by a RAID level 1 organization.

**11.23** Assume that you have a mixed configuration comprising disks organized as RAID level 1 and RAID level 5 disks. Assume that the system has flexibility in deciding which disk organization to use for storing a particular file. Which files should be stored in the RAID level 1 disks and which in the RAID level 5 disks in order to optimize performance?

**11.24** The reliability of a storage device is typically described in terms of mean time between failures (MTBF). Although this quantity is called a "time," the MTBF actually is measured in drive-hours per failure.

a. If a system contains 1,000 disk drives, each of which has a 750,000-hour MTBF, which of the following best describes how often a drive failure will occur in that disk farm: once per thousand years, once per century, once per decade, once per year, once per month, once per week, once per day, once per hour, once per minute, or once per second?

b. Mortality statistics indicate that, on the average, a U.S. resident has about 1 chance in 1,000 of dying between the ages of 20 and 21. Deduce the MTBF hours for 20-year-olds. Convert this figure from hours to years. What does this MTBF tell you about the expected lifetime of a 20-year-old?

c. The manufacturer guarantees a 1-million-hour MTBF for a certain model of disk drive. What can you conclude about the number of years for which one of these drives is under warranty?

**11.25** Discuss the relative advantages and disadvantages of sector sparing and sector slipping.

**11.26** Discuss the reasons why the operating system might require accurate information on how blocks are stored on a disk. How could the operating system improve file-system performance with this knowledge?

## Programming Problems

**11.27** Write a program that implements the following disk-scheduling algorithms:

a. FCFS

b. SCAN

c. C-SCAN

Your program will service a disk with 5,000 cylinders numbered 0 to 4,999. The program will generate a random series of 1,000 cylinder requests and service them according to each of the algorithms listed above. The program will be passed the initial position of the disk head (as a parameter on the command line) and report the total amount of head movement required by each algorithm.

CHAPTER 12

# I/O Systems

The two main jobs of a computer are I/O and computing. In many cases, the main job is I/O, and the computing or processing is merely incidental. For instance, when we browse a web page or edit a file, our immediate interest is to read or enter some information, not to compute an answer.

The role of the operating system in computer I/O is to manage and control I/O operations and I/O devices. Although related topics appear in other chapters, here we bring together the pieces to paint a complete picture of I/O. First, we describe the basics of I/O hardware, because the nature of the hardware interface places constraints on the internal facilities of the operating system. Next, we discuss the I/O services provided by the operating system and the embodiment of these services in the application I/O interface. Then, we explain how the operating system bridges the gap between the hardware interface and the application interface. We also discuss the UNIX System V STREAMS mechanism, which enables an application to assemble pipelines of driver code dynamically. Finally, we discuss the performance aspects of I/O and the principles of operating-system design that improve I/O performance.

### CHAPTER OBJECTIVES

- Explore the structure of an operating system's I/O subsystem.
- Discuss the principles and complexities of I/O hardware.
- Explain the performance aspects of I/O hardware and software.

## 12.1 Overview

The control of devices connected to the computer is a major concern of operating-system designers. Because I/O devices vary so widely in their function and speed (consider a mouse, a hard disk, a flash drive, and a tape robot), varied methods are needed to control them. These methods form the I/O subsystem of the kernel, which separates the rest of the kernel from the complexities of managing I/O devices.

I/O-device technology exhibits two conflicting trends. On the one hand, we see increasing standardization of software and hardware interfaces. This trend helps us to incorporate improved device generations into existing computers and operating systems. On the other hand, we see an increasingly broad variety of I/O devices. Some new devices are so unlike previous devices that it is a challenge to incorporate them into our computers and operating systems. This challenge is met by a combination of hardware and software techniques. The basic I/O hardware elements, such as ports, buses, and device controllers, accommodate a wide variety of I/O devices. To encapsulate the details and oddities of different devices, the kernel of an operating system is structured to use device-driver modules. The **device drivers** present a uniform device-access interface to the I/O subsystem, much as system calls provide a standard interface between the application and the operating system.

## 12.2 I/O Hardware

Computers operate a great many kinds of devices. Most fit into the general categories of storage devices (disks, tapes), transmission devices (network connections, Bluetooth), and human-interface devices (screen, keyboard, mouse, audio in and out). Other devices are more specialized, such as those involved in the steering of a jet. In these aircraft, a human gives input to the flight computer via a joystick and foot pedals, and the computer sends output commands that cause motors to move rudders and flaps and fuels to the engines. Despite the incredible variety of I/O devices, though, we need only a few concepts to understand how the devices are attached and how the software can control the hardware.

A device communicates with a computer system by sending signals over a cable or even through the air. The device communicates with the machine via a connection point, or **port**—for example, a serial port. (The term **PHY**, shorthand for the OSI model physical layer, is also used in reference to ports but is more common in data-center nomenclature.) If devices share a common set of wires, the connection is called a bus. A **bus**, like the PCI bus used in most computers today, is a set of wires and a rigidly defined protocol that specifies a set of messages that can be sent on the wires. In terms of the electronics, the messages are conveyed by patterns of electrical voltages applied to the wires with defined timings. When device *A* has a cable that plugs into device *B*, and device *B* has a cable that plugs into device *C*, and device *C* plugs into a port on the computer, this arrangement is called a **daisy chain**. A daisy chain usually operates as a bus.

Buses are used widely in computer architecture and vary in their signaling methods, speed, throughput, and connection methods. A typical PC bus structure appears in Figure 12.1. In the figure, a **PCIe bus** (the common PC system bus) connects the processor–memory subsystem to fast devices, and an **expansion bus** connects relatively slow devices, such as the keyboard and serial and USB ports. In the lower-left portion of the figure, four disks are connected together on a **serial-attached SCSI** (**SAS**) bus plugged into an SAS controller. PCIe is a flexible bus that sends data over one or more "lanes." A lane is composed of two signaling pairs, one pair for receiving data and the other for transmitting. Each lane is therefore composed of four wires, and each

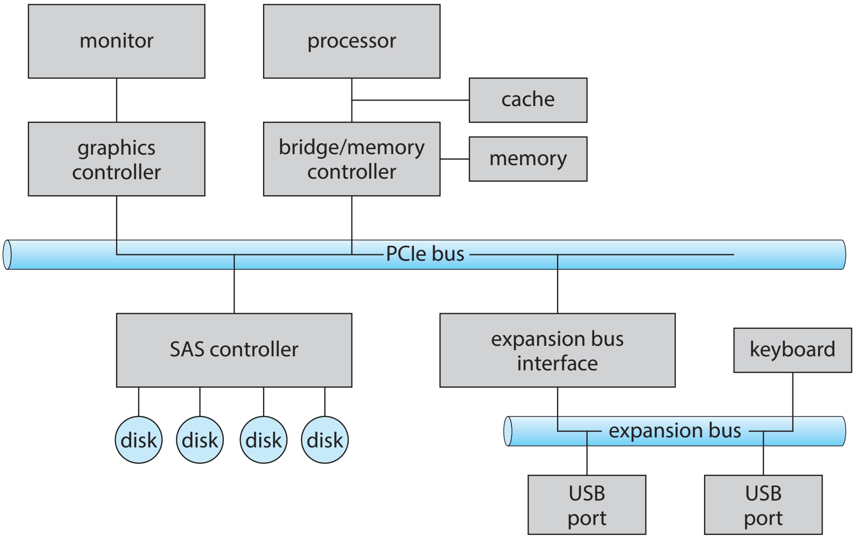


**Figure 12.1** A typical PC bus structure.

lane is used as a full-duplex byte stream, transporting data packets in an eight-bit byte format simultaneously in both directions. Physically, PCIe links may contain 1, 2, 4, 8, 12, 16, or 32 lanes, as signified by an “x” prefix. A PCIe card or connector that uses 8 lanes is designated x8, for example. In addition, PCIe has gone through multiple “generations,” with more coming in the future. Thus, for example, a card might be “PCIe gen3 x8”, which means it works with generation 3 of PCIe and uses 8 lanes. Such a device has maximum throughput of 8 gigabytes per second. Details about PCIe can be found at https://pcisig.com.

A **controller** is a collection of electronics that can operate a port, a bus, or a device. A serial-port controller is a simple device controller. It is a single chip (or portion of a chip) in the computer that controls the signals on the wires of a serial port. By contrast, a **fibre channel** (**FC**) bus controller is not simple. Because the FC protocol is complex and used in data centers rather than on PCs, the FC bus controller is often implemented as a separate circuit board —or a **host bus adapter** (**HBA**)—that connects to a bus in the computer. It typically contains a processor, microcode, and some private memory to enable it to process the FC protocol messages. Some devices have their own built-in controllers. If you look at a disk drive, you will see a circuit board attached to one side. This board is the disk controller. It implements the disk side of the protocol for some kinds of connection—SAS and SATA, for instance. It has microcode and a processor to do many tasks, such as bad-sector mapping, prefetching, buffering, and caching.

### 12.2.1 Memory-Mapped I/O

How does the processor give commands and data to a controller to accomplish an I/O transfer? The short answer is that the controller has one or more registers for data and control signals. The processor communicates with the controller by reading and writing bit patterns in these registers. One way in which this communication can occur is through the use of special I/O instructions

| I/O address range (hexadecimal) | device |
|---|---|
| 000–00F | DMA controller |
| 020–021 | interrupt controller |
| 040–043 | timer |
| 200–20F | game controller |
| 2F8–2FF | serial port (secondary) |
| 320–32F | hard-disk controller |
| 378–37F | parallel port |
| 3D0–3DF | graphics controller |
| 3F0–3F7 | diskette-drive controller |
| 3F8–3FF | serial port (primary) |

**Figure 12.2** Device I/O port locations on PCs (partial).

that specify the transfer of a byte or a word to an I/O port address. The I/O instruction triggers bus lines to select the proper device and to move bits into or out of a device register. Alternatively, the device can support **memory-mapped I/O**. In this case, the device-control registers are mapped into the address space of the processor. The CPU executes I/O requests using the standard data-transfer instructions to read and write the device-control registers at their mapped locations in physical memory.

In the past, PCs often used I/O instructions to control some devices and memory-mapped I/O to control others. Figure 12.2 shows the usual I/O port addresses for PCs. The graphics controller has I/O ports for basic control operations, but the controller has a large memory-mapped region to hold screen contents. A thread sends output to the screen by writing data into the memory-mapped region. The controller generates the screen image based on the contents of this memory. This technique is simple to use. Moreover, writing millions of bytes to the graphics memory is faster than issuing millions of I/O instructions. Therefore, over time, systems have moved toward memory-mapped I/O. Today, most I/O is performed by device controllers using memory-mapped I/O.

I/O device control typically consists of four registers, called the status, control, data-in, and data-out registers.

- The **data-in register** is read by the host to get input.
- The **data-out register** is written by the host to send output.
- The **status register** contains bits that can be read by the host. These bits indicate states, such as whether the current command has completed, whether a byte is available to be read from the data-in register, and whether a device error has occurred.
- The **control register** can be written by the host to start a command or to change the mode of a device. For instance, a certain bit in the control register of a serial port chooses between full-duplex and half-duplex com-

munication, another bit enables parity checking, a third bit sets the word length to 7 or 8 bits, and other bits select one of the speeds supported by the serial port.

The data registers are typically 1 to 4 bytes in size. Some controllers have FIFO chips that can hold several bytes of input or output data to expand the capacity of the controller beyond the size of the data register. A FIFO chip can hold a small burst of data until the device or host is able to receive those data.

### 12.2.2 Polling

The complete protocol for interaction between the host and a controller can be intricate, but the basic handshaking notion is simple. We explain handshaking with an example. Assume that 2 bits are used to coordinate the producer–consumer relationship between the controller and the host. The controller indicates its state through the `busy` bit in the `status` register. (Recall that to ***set*** a bit means to write a 1 into the bit and to ***clear*** a bit means to write a 0 into it.) The controller sets the `busy` bit when it is busy working and clears the `busy` bit when it is ready to accept the next command. The host signals its wishes via the `command-ready` bit in the `command` register. The host sets the `command-ready` bit when a command is available for the controller to execute. For this example, the host writes output through a port, coordinating with the controller by handshaking as follows.

1. The host repeatedly reads the `busy` bit until that bit becomes clear.
2. The host sets the `write` bit in the `command` register and writes a byte into the `data-out` register.
3. The host sets the `command-ready` bit.
4. When the controller notices that the `command-ready` bit is set, it sets the `busy` bit.
5. The controller reads the command register and sees the `write` command. It reads the `data-out` register to get the byte and does the I/O to the device.
6. The controller clears the `command-ready` bit, clears the `error` bit in the status register to indicate that the device I/O succeeded, and clears the `busy` bit to indicate that it is finished.

This loop is repeated for each byte.

In step 1, the host is **busy-waiting** or **polling**: it is in a loop, reading the `status` register over and over until the `busy` bit becomes clear. If the controller and device are fast, this method is a reasonable one. But if the wait may be long, the host should probably switch to another task. How, then, does the host know when the controller has become idle? For some devices, the host must service the device quickly, or data will be lost. For instance, when data are streaming in on a serial port or from a keyboard, the small buffer on the controller will overflow and data will be lost if the host waits too long before returning to read the bytes.

In many computer architectures, three CPU-instruction cycles are sufficient to poll a device: `read` a device register, `logical-and` to extract a status bit, and `branch` if not zero. Clearly, the basic polling operation is efficient. But polling becomes inefficient when it is attempted repeatedly yet rarely finds a device ready for service, while other useful CPU processing remains undone. In such instances, it may be more efficient to arrange for the hardware controller to notify the CPU when the device becomes ready for service, rather than to require the CPU to poll repeatedly for an I/O completion. The hardware mechanism that enables a device to notify the CPU is called an **interrupt**.

### 12.2.3 Interrupts

The basic interrupt mechanism works as follows. The CPU hardware has a wire called the **interrupt-request line** that the CPU senses after executing every instruction. When the CPU detects that a controller has asserted a signal on the interrupt-request line, the CPU performs a state save and jumps to the **interrupt-handler routine** at a fixed address in memory. The interrupt handler determines the cause of the interrupt, performs the necessary processing, performs a state restore, and executes a `return from interrupt` instruction to return the CPU to the execution state prior to the interrupt. We say that the device controller ***raises*** an interrupt by asserting a signal on the interrupt request line, the CPU ***catches*** the interrupt and ***dispatches*** it to the interrupt

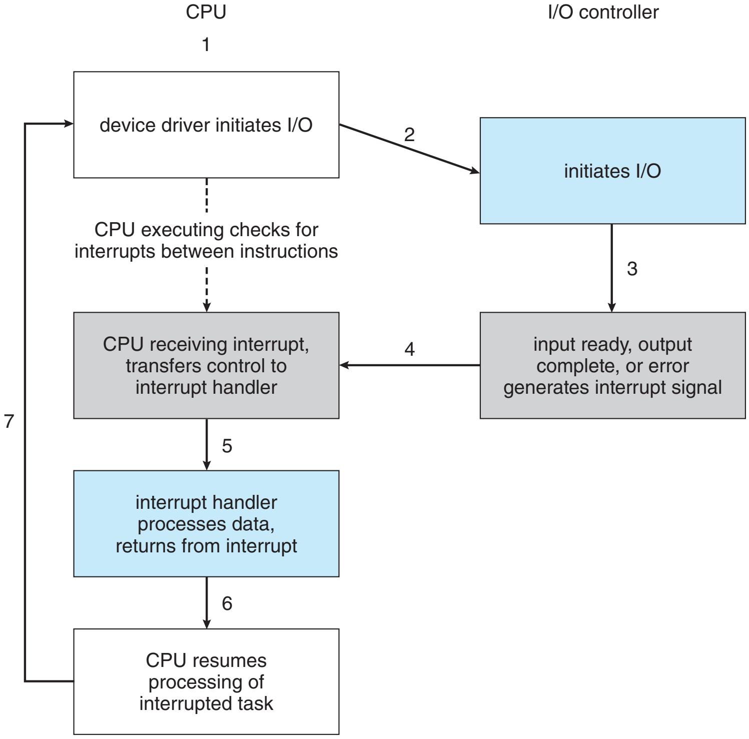
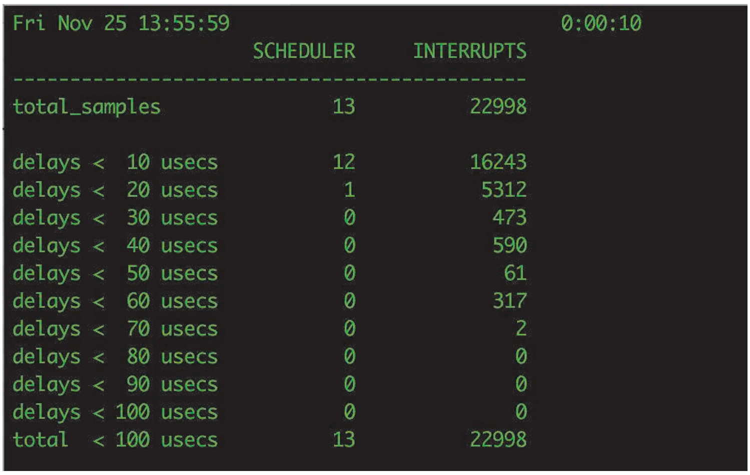

CPU
I/O controller
1
device driver initiates I/O
2
initiates I/O
CPU executing checks for interrupts between instructions
3
CPU receiving interrupt, transfers control to interrupt handler
4
input ready, output complete, or error generates interrupt signal
5
interrupt handler processes data, returns from interrupt
6
CPU resumes processing of interrupted task
7

**Figure 12.3** Interrupt-driven I/O cycle.



```
Fri Nov 25 13:55:59                             0:00:10
                     SCHEDULER     INTERRUPTS
---------------------------------------------
total_samples               13          22998

delays <  10 usecs          12          16243
delays <  20 usecs           1           5312
delays <  30 usecs           0            473
delays <  40 usecs           0            590
delays <  50 usecs           0             61
delays <  60 usecs           0            317
delays <  70 usecs           0              2
delays <  80 usecs           0              0
delays <  90 usecs           0              0
delays < 100 usecs           0              0
total  < 100 usecs          13          22998
```

**Figure 12.4** Latency command on Mac OS X.

handler, and the handler ***clears*** the interrupt by servicing the device. Figure 12.3 summarizes the interrupt-driven I/O cycle.

We stress interrupt management in this chapter because even single-user modern systems manage hundreds of interrupts per second and servers hundreds of thousands per second. For example, Figure 12.4 shows the `latency` command output on macOS, revealing that over ten seconds a quiet desktop computer performed almost 23,000 interrupts.

The basic interrupt mechanism just described enables the CPU to respond to an asynchronous event, as when a device controller becomes ready for service. In a modern operating system, however, we need more sophisticated interrupt-handling features.

1. We need the ability to defer interrupt handling during critical processing.
2. We need an efficient way to dispatch to the proper interrupt handler for a device without first polling all the devices to see which one raised the interrupt.
3. We need multilevel interrupts, so that the operating system can distinguish between high- and low-priority interrupts and can respond with the appropriate degree of urgency when there are multiple concurrent interrupts.
4. We need a way for an instruction to get the operating system's attention directly (separately from I/O requests), for activities such as page faults and errors such as division by zero. As we shall see, this task is accomplished by "traps."

In modern computer hardware, these features are provided by the CPU and by the **interrupt-controller hardware**.

Most CPUs have two interrupt request lines. One is the **nonmaskable interrupt**, which is reserved for events such as unrecoverable memory errors. The second interrupt line is **maskable**: it can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted. The maskable interrupt is used by device controllers to request service.

The interrupt mechanism accepts an **address**—a number that selects a specific interrupt-handling routine from a small set. In most architectures, this address is an offset in a table called the **interrupt vector**. This vector contains the memory addresses of specialized interrupt handlers. The purpose of a vectored interrupt mechanism is to reduce the need for a single interrupt handler to search all possible sources of interrupts to determine which one needs service. In practice, however, computers have more devices (and, hence, interrupt handlers) than they have address elements in the interrupt vector. A common way to solve this problem is to use **interrupt chaining**, in which each element in the interrupt vector points to the head of a list of interrupt handlers. When an interrupt is raised, the handlers on the corresponding list are called one by one, until one is found that can service the request. This structure is a compromise between the overhead of a huge interrupt table and the inefficiency of dispatching to a single interrupt handler.

Figure 12.5 illustrates the design of the interrupt vector for the Intel Pentium processor. The events from 0 to 31, which are nonmaskable, are used to signal various error conditions (which cause system crashes), page faults (needing immediate action), and debugging requests (stopping normal operation and jumping to a debugger application). The events from 32 to 255, which are maskable, are used for purposes such as device-generated interrupts.

| vector number | description |
|---|---|
| 0 | divide error |
| 1 | debug exception |
| 2 | null interrupt |
| 3 | breakpoint |
| 4 | INTO-detected overflow |
| 5 | bound range exception |
| 6 | invalid opcode |
| 7 | device not available |
| 8 | double fault |
| 9 | coprocessor segment overrun (reserved) |
| 10 | invalid task state segment |
| 11 | segment not present |
| 12 | stack fault |
| 13 | general protection |
| 14 | page fault |
| 15 | (Intel reserved, do not use) |
| 16 | floating-point error |
| 17 | alignment check |
| 18 | machine check |
| 19–31 | (Intel reserved, do not use) |
| 32–255 | maskable interrupts |

**Figure 12.5** Intel Pentium processor event-vector table.

The interrupt mechanism also implements a system of **interrupt priority levels**. These levels enable the CPU to defer the handling of low-priority interrupts without masking all interrupts and make it possible for a high-priority interrupt to preempt the execution of a low-priority interrupt.

A modern operating system interacts with the interrupt mechanism in several ways. At boot time, the operating system probes the hardware buses to determine what devices are present and installs the corresponding interrupt handlers into the interrupt vector. During I/O, the various device controllers raise interrupts when they are ready for service. These interrupts signify that output has completed, or that input data are available, or that a failure has been detected. The interrupt mechanism is also used to handle a wide variety of **exceptions**, such as dividing by zero, accessing a protected or nonexistent memory address, or attempting to execute a privileged instruction from user mode. The events that trigger interrupts have a common property: they are occurrences that induce the operating system to execute an urgent, self-contained routine.

Because interrupt handing in many cases is time and resource constrained and therefore complicated to implement, systems frequently split interrupt management between a **first-leve interrupt handler** (**FLIH**) and a **second-level interrupt handler** (**SLIH**). The FLIH performs the context switch, state storage, and queuing of a handling operation, while the separately scheduled SLIH performs the handling of the requested operation.

Operating systems have other good uses for interrupts as well. For example, many operating systems use the interrupt mechanism for virtual memory paging. A page fault is an exception that raises an interrupt. The interrupt suspends the current process and jumps to the page-fault handler in the kernel. This handler saves the state of the process, moves the process to the wait queue, performs page-cache management, schedules an I/O operation to fetch the page, schedules another process to resume execution, and then returns from the interrupt.

Another example is found in the implementation of system calls. Usually, a program uses library calls to issue system calls. The library routines check the arguments given by the application, build a data structure to convey the arguments to the kernel, and then execute a special instruction called a **software interrupt**, or **trap**. This instruction has an operand that identifies the desired kernel service. When a process executes the trap instruction, the interrupt hardware saves the state of the user code, switches to kernel mode, and dispatches to the kernel routine or thread that implements the requested service. The trap is given a relatively low interrupt priority compared with those assigned to device interrupts—executing a system call on behalf of an application is less urgent than servicing a device controller before its FIFO queue overflows and loses data.

Interrupts can also be used to manage the flow of control within the kernel. For example, consider the case of the processing required to complete a disk read. One step may copy data from kernel space to the user buffer. This copying is time consuming but not urgent—it should not block other high-priority interrupt handling. Another step is to start the next pending I/O for that disk drive. This step has higher priority. If the disks are to be used efficiently, we need to start the next I/O as soon as the previous one completes. Consequently, a pair of interrupt handlers implements the kernel code that completes a disk read. The high-priority handler records the I/O status, clears the device interrupt, starts the next pending I/O, and raises a low-priority interrupt to complete the work. Later, when the CPU is not occupied with high-priority work, the low-priority interrupt will be dispatched. The corresponding handler completes the user-level I/O by copying data from kernel buffers to the application space and then calling the scheduler to place the application on the ready queue.

A threaded kernel architecture is well suited to implement multiple interrupt priorities and to enforce the precedence of interrupt handling over background processing in kernel and application routines. We illustrate this point with the Solaris kernel. In Solaris, interrupt handlers are executed as kernel threads. A range of high scheduling priorities is reserved for these threads. These priorities give interrupt handlers precedence over application code and kernel housekeeping and implement the priority relationships among interrupt handlers. The priorities cause the Solaris thread scheduler to preempt low-priority interrupt handlers in favor of higher-priority ones, and the threaded implementation enables multiprocessor hardware to run several interrupt handlers concurrently. We describe the interrupt architecture of Linux in Chapter 20, Windows10 in Chapter 21, and UNIX in Appendix C.

In summary, interrupts are used throughout modern operating systems to handle asynchronous events and to trap to supervisor-mode routines in the kernel. To enable the most urgent work to be done first, modern computers use a system of interrupt priorities. Device controllers, hardware faults, and system calls all raise interrupts to trigger kernel routines. Because interrupts are used so heavily for time-sensitive processing, efficient interrupt handling is required for good system performance. Interrupt-driven I/O is now much more common than polling, with polling being used for high-throughput I/O. Sometimes the two are used together. Some device drivers use interrupts when the I/O rate is low and switch to polling when the rate increases to the point where polling is faster and more efficient.

### 12.2.4 Direct Memory Access

For a device that does large transfers, such as a disk drive, it seems wasteful to use an expensive general-purpose processor to watch status bits and to feed data into a controller register one byte at a time—a process termed **programmed I/O** (**PIO**). Computers avoid burdening the main CPU with PIO by offloading some of this work to a special-purpose processor called a **direct-memory-access** (**DMA**) controller. To initiate a DMA transfer, the host writes a DMA command block into memory. This block contains a pointer to the source of a transfer, a pointer to the destination of the transfer, and a count of the number of bytes to be transferred. A command block can be more complex, including a list of sources and destinations addresses that are not contiguous. This **scatter–gather** method allows multiple transfers to be executed via a single DMA command. The CPU writes the address of this command block to the DMA controller, then goes on with other work. The DMA controller proceeds to operate the memory bus directly, placing addresses on the bus to perform transfers without the help of the main CPU. A simple DMA controller is a standard component in all modern computers, from smartphones to mainframes.

Note that it is most straightforward for the target address to be in kernel address space. If it were in user space, the user could, for example, modify the contents of that space during the transfer, losing some set of data. To get the DMA-transferred data to the user space for thread access, however, a second copy operation, this time from kernel memory to user memory, is needed. This **double buffering** is inefficient. Over time, operating systems have moved to using memory-mapping (see Section 12.2.1) to perform I/O transfers directly between devices and user address space.

Handshaking between the DMA controller and the device controller is performed via a pair of wires called **DMA-request** and **DMA-acknowledge**. The device controller places a signal on the DMA-request wire when a word of data is available for transfer. This signal causes the DMA controller to seize the memory bus, place the desired address on the memory-address wire, and place a signal on the DMA-acknowledge wire. When the device controller receives the DMA-acknowledge signal, it transfers the word of data to memory and removes the DMA-request signal.

When the entire transfer is finished, the DMA controller interrupts the CPU. This process is depicted in Figure 12.6. When the DMA controller seizes the memory bus, the CPU is momentarily prevented from accessing main memory, although it can still access data items in its caches. Although this **cycle stealing** can slow down the CPU computation, offloading the data-transfer work to a DMA controller generally improves the total system performance. Some computer architectures use physical memory addresses for DMA, but

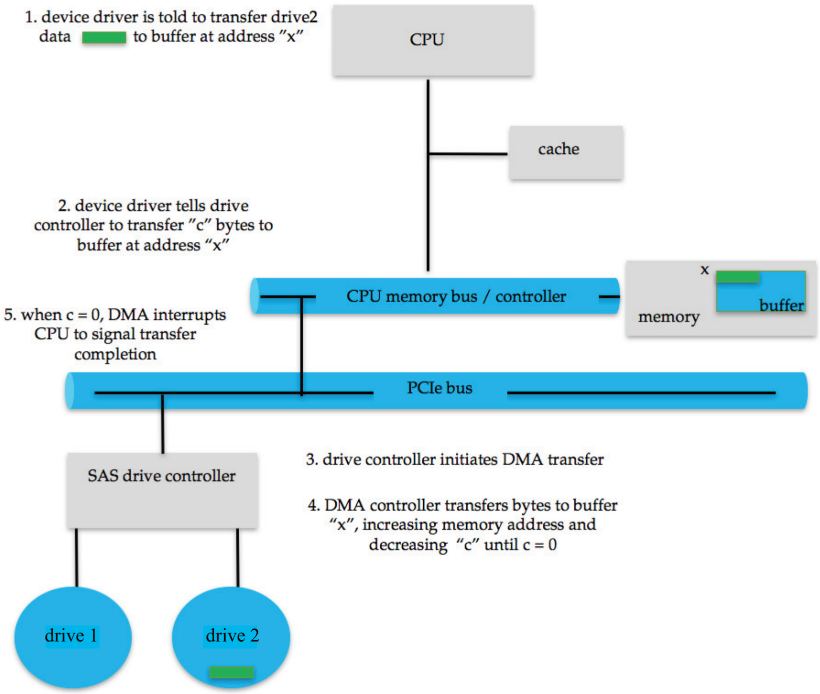

1. device driver is told to transfer drive2 data to buffer at address "x"
CPU
cache
2. device driver tells drive controller to transfer "c" bytes to buffer at address "x"
5. when c = 0, DMA interrupts CPU to signal transfer completion
CPU memory bus / controller
x
buffer
memory
PCIe bus
SAS drive controller
3. drive controller initiates DMA transfer
4. DMA controller transfers bytes to buffer "x", increasing memory address and decreasing "c" until c = 0
drive 1
drive 2

**Figure 12.6** Steps in a DMA transfer.

others perform **direct virtual memory access** (**DVMA**), using virtual addresses that undergo translation to physical addresses. DVMA can perform a transfer between two memory-mapped devices without the intervention of the CPU or the use of main memory.

On protected-mode kernels, the operating system generally prevents processes from issuing device commands directly. This discipline protects data from access-control violations and also protects the system from erroneous use of device controllers, which could cause a system crash. Instead, the operating system exports functions that a sufficiently privileged process can use to access low-level operations on the underlying hardware. On kernels without memory protection, processes can access device controllers directly. This direct access can be used to achieve high performance, since it can avoid kernel communication, context switches, and layers of kernel software. Unfortunately, it interferes with system security and stability. Common general-purpose operating systems protect memory and devices so that the system can try to guard against erroneous or malicious applications.

### 12.2.5 I/O Hardware Summary

Although the hardware aspects of I/O are complex when considered at the level of detail of electronics-hardware design, the concepts that we have just described are sufficient to enable us to understand many I/O features of operating systems. Let's review the main concepts:

- A bus
- A controller
- An I/O port and its registers
- The handshaking relationship between the host and a device controller
- The execution of this handshaking in a polling loop or via interrupts
- The offloading of this work to a DMA controller for large transfers

We gave a basic example of the handshaking that takes place between a device controller and the host earlier in this section. In reality, the wide variety of available devices poses a problem for operating-system implementers. Each kind of device has its own set of capabilities, control-bit definitions, and protocols for interacting with the host—and they are all different. How can the operating system be designed so that we can attach new devices to the computer without rewriting the operating system? And when the devices vary so widely, how can the operating system give a convenient, uniform I/O interface to applications? We address those questions next.

## 12.3 Application I/O Interface

In this section, we discuss structuring techniques and interfaces for the operating system that enable I/O devices to be treated in a standard, uniform way. We explain, for instance, how an application can open a file on a disk without

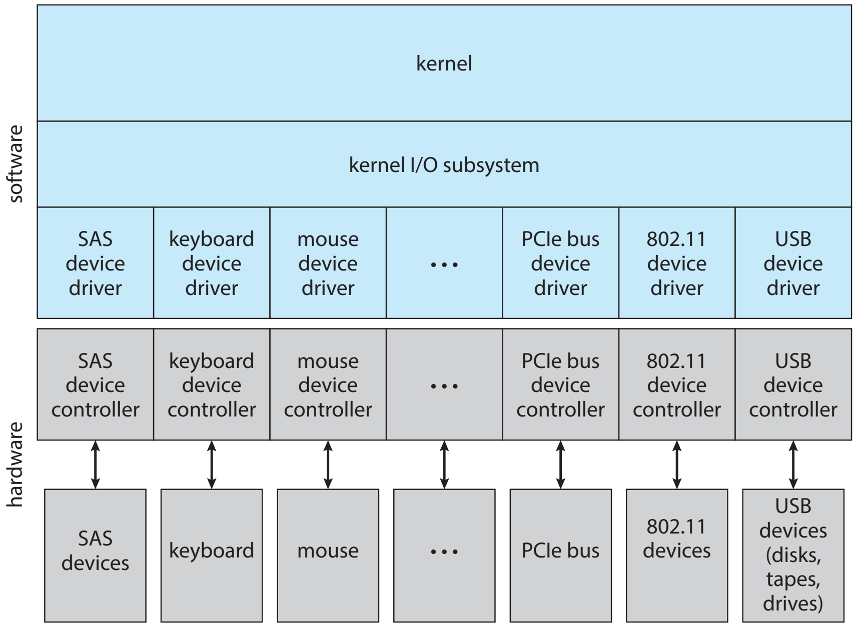

software
hardware
kernel
kernel I/O subsystem
SAS device driver
keyboard device driver
mouse device driver
• • •
PCIe bus device driver
802.11 device driver
USB device driver
SAS device controller
keyboard device controller
mouse device controller
• • •
PCIe bus device controller
802.11 device controller
USB device controller
SAS devices
keyboard
mouse
• • •
PCIe bus
802.11 devices
USB devices (disks, tapes, drives)

**Figure 12.7** A kernel I/O structure.

knowing what kind of disk it is and how new disks and other devices can be added to a computer without disruption of the operating system.

Like other complex software-engineering problems, the approach here involves abstraction, encapsulation, and software layering. Specifically, we can abstract away the detailed differences in I/O devices by identifying a few general kinds. Each general kind is accessed through a standardized set of functions—an **interface**. The differences are encapsulated in kernel modules called device drivers that internally are custom-tailored to specific devices but that export one of the standard interfaces. Figure 12.7 illustrates how the I/O-related portions of the kernel are structured in software layers.

The purpose of the device-driver layer is to hide the differences among device controllers from the I/O subsystem of the kernel, much as the I/O system calls encapsulate the behavior of devices in a few generic classes that hide hardware differences from applications. Making the I/O subsystem independent of the hardware simplifies the job of the operating-system developer. It also benefits the hardware manufacturers. They either design new devices to be compatible with an existing host controller interface (such as SATA), or they write device drivers to interface the new hardware to popular operating systems. Thus, we can attach new peripherals to a computer without waiting for the operating-system vendor to develop support code.

Unfortunately for device-hardware manufacturers, each type of operating system has its own standards for the device-driver interface. A given device may ship with multiple device drivers—for instance, drivers for Windows, Linux, AIX, and macOS. Devices vary on many dimensions, as illustrated in Figure 12.8.

| aspect | variation | example |
|---|---|---|
| data-transfer mode | character<br>block | terminal<br>disk |
| access method | sequential<br>random | modem<br>CD-ROM |
| transfer schedule | synchronous<br>asynchronous | tape<br>keyboard |
| sharing | dedicated<br>sharable | tape<br>keyboard |
| device speed | latency<br>seek time<br>transfer rate<br>delay between operations | |
| I/O direction | read only<br>write only<br>read–write | CD-ROM<br>graphics controller<br>disk |

**Figure 12.8** Characteristics of I/O devices.

- **Character-stream or block**. A character-stream device transfers bytes one by one, whereas a block device transfers a block of bytes as a unit.
- **Sequential or random access**. A sequential device transfers data in a fixed order determined by the device, whereas the user of a random-access device can instruct the device to seek to any of the available data storage locations.
- **Synchronous or asynchronous**. A synchronous device performs data transfers with predictable response times, in coordination with other aspects of the system. An asynchronous device exhibits irregular or unpredictable response times not coordinated with other computer events.
- **Sharable or dedicated**. A sharable device can be used concurrently by several processes or threads; a dedicated device cannot.
- **Speed of operation**. Device speeds range from a few bytes per second to gigabytes per second.
- **Read–write, read only, write once**. Some devices perform both input and output, but others support only one data transfer direction. Some allow data to be modified after write, but others can be written only once and are read-only thereafter.

For the purpose of application access, many of these differences are hidden by the operating system, and the devices are grouped into a few conventional types. The resulting styles of device access have been found to be useful and broadly applicable. Although the exact system calls may differ across operating systems, the device categories are fairly standard. The major access conventions include block I/O, character-stream I/O, memory-mapped file access, and network sockets. Operating systems also provide special system calls to access a few additional devices, such as a time-of-day clock and a timer. Some operating systems provide a set of system calls for graphical display, video, and audio devices.

Most operating systems also have an **escape** (or **back door**) that transparently passes arbitrary commands from an application to a device driver. In UNIX, this system call is `ioctl()` (for "I/O control"). The `ioctl()` system call enables an application to access any functionality that can be implemented by any device driver, without the need to invent a new system call. The `ioctl()` system call has three arguments. The first is a device identifier that connects the application to the driver by referring to a hardware device managed by that driver. The second is an integer that selects one of the commands implemented in the driver. The third is a pointer to an arbitrary data structure in memory that enables the application and driver to communicate any necessary control information or data.

The device identifier in UNIX and Linux is a tuple of "major and minor" device numbers. The major number is the device type, and the second is the instance of that device. For example, consider these SSD devices on a system. If one issues a command:

```
% ls -l /dev/sda*
```

then the following output

```
brw-rw---- 1 root disk 8, 0 Mar 16 09:18 /dev/sda
brw-rw---- 1 root disk 8, 1 Mar 16 09:18 /dev/sda1
brw-rw---- 1 root disk 8, 2 Mar 16 09:18 /dev/sda2
brw-rw---- 1 root disk 8, 3 Mar 16 09:18 /dev/sda3
```

shows that 8 is the major device number. The operating system uses that information to route I/O requests to the appropriate device driver. The minor numbers 0, 1, 2, and 3 indicate the instance of the device, allowing requests for I/O to a device entry to select the exact device for the request.

### 12.3.1 Block and Character Devices

The **block-device interface** captures all the aspects necessary for accessing disk drives and other block-oriented devices. The device is expected to understand commands such as `read()` and `write()`. If it is a random-access device, it is also expected to have a `seek()` command to specify which block to transfer next. Applications normally access such a device through a file-system interface. We can see that `read()`, `write()`, and `seek()` capture the essential behaviors of block-storage devices, so that applications are insulated from the low-level differences among those devices.

The operating system itself, as well as special applications such as database-management systems, may prefer to access a block device as a simple linear array of blocks. This mode of access is sometimes called **raw I/O**. If the application performs its own buffering, then using a file system would cause extra, unneeded buffering. Likewise, if an application provides its own locking of blocks or regions, then any operating-system locking services would be redundant at the least and contradictory at the worst. To avoid these conflicts, raw-device access passes control of the device directly to the application, letting the operating system step out of the way. Unfortunately, no operating-system services are then performed on this device. A compromise that is becoming common is for the operating system to allow a mode of operation on a file that disables buffering and locking. In the UNIX world, this is called **direct I/O**.

Memory-mapped file access can be layered on top of block-device drivers. Rather than offering read and write operations, a memory-mapped interface provides access to disk storage via an array of bytes in main memory. The system call that maps a file into memory returns the virtual memory address that contains a copy of the file. The actual data transfers are performed only when needed to satisfy access to the memory image. Because the transfers are handled by the same mechanism as that used for demand-paged virtual memory access, memory-mapped I/O is efficient. Memory mapping is also convenient for programmers—access to a memory-mapped file is as simple as reading from and writing to memory. Operating systems that offer virtual memory commonly use the mapping interface for kernel services. For instance, to execute a program, the operating system maps the executable into memory and then transfers control to the entry address of the executable. The mapping interface is also commonly used for kernel access to swap space on disk.

A keyboard is an example of a device that is accessed through a **character-stream interface**. The basic system calls in this interface enable an application to `get()` or `put()` one character. On top of this interface, libraries can be built that offer line-at-a-time access, with buffering and editing services (for example, when a user types a backspace, the preceding character is removed from the input stream). This style of access is convenient for input devices such as keyboards, mice, and modems that produce data for input “spontaneously” —that is, at times that cannot necessarily be predicted by the application. This access style is also good for output devices such as printers and audio boards, which naturally fit the concept of a linear stream of bytes.

### 12.3.2 Network Devices

Because the performance and addressing characteristics of network I/O differ significantly from those of disk I/O, most operating systems provide a network I/O interface that is different from the `read()-write()-seek()` interface used for disks. One interface available in many operating systems, including UNIX and Windows, is the network **socket** interface.

Think of a wall socket for electricity: any electrical appliance can be plugged in. By analogy, the system calls in the socket interface enable an application to create a socket, to connect a local socket to a remote address (which plugs this application into a socket created by another application), to listen for any remote application to plug into the local socket, and to send and receive packets over the connection. To support the implementation of network servers, the socket interface also provides a function called `select()` that manages a set of sockets. A call to `select()` returns information about which sockets have a packet waiting to be received and which sockets have room to accept a packet to be sent. The use of `select()` eliminates the polling and busy waiting that would otherwise be necessary for network I/O. These functions encapsulate the essential behaviors of networks, greatly facilitating the creation of distributed applications that can use any underlying network hardware and protocol stack.

Many other approaches to interprocess communication and network communication have been implemented. For instance, Windows provides one interface to the network interface card and a second interface to the network protocols. In UNIX, which has a long history as a proving ground for network technology, we find half-duplex pipes, full-duplex FIFOs, full-duplex STREAMS, message queues, and sockets. Information on UNIX networking is given in Section C.9.

### 12.3.3 Clocks and Timers

Most computers have hardware clocks and timers that provide three basic functions:

- Give the current time.
- Give the elapsed time.
- Set a timer to trigger operation *X* at time *T*.

These functions are used heavily by the operating system, as well as by time-sensitive applications. Unfortunately, the system calls that implement these functions are not standardized across operating systems.

The hardware to measure elapsed time and to trigger operations is called a **programmable interval timer**. It can be set to wait a certain amount of time and then generate an interrupt, and it can be set to do this once or to repeat the process to generate periodic interrupts. The scheduler uses this mechanism to generate an interrupt that will preempt a process at the end of its time slice. The disk I/O subsystem uses it to invoke the periodic flushing of dirty cache buffers to disk, and the network subsystem uses it to cancel operations that are proceeding too slowly because of network congestion or failures. The operating system may also provide an interface for user processes to use timers. The operating system can support more timer requests than the number of timer hardware channels by simulating virtual clocks. To do so, the kernel (or the timer device driver) maintains a list of interrupts wanted by its own routines and by user requests, sorted in earliest-time-first order. It sets the timer for the earliest time. When the timer interrupts, the kernel signals the requester and reloads the timer with the next earliest time.

Computers have clock hardware that is used for a variety of purposes. Modern PCs include a **high-performance event timer** (**HPET**), which runs at rates in the 10-megahertz range. It has several comparators that can be set to trigger once or repeatedly when the value they hold matches that of the HPET. The trigger generates an interrupt, and the operating system's clock management routines determine what the timer was for and what action to take. The precision of triggers is limited by the resolution of the timer, together with the overhead of maintaining virtual clocks. Furthermore, if the timer ticks are used to maintain the system time-of-day clock, the system clock can drift. Drift can be corrected via protocols designed for that purpose, such as NTP, the **network time protocol**, which uses sophisticated latency calculations to keep a computer's clock accurate almost to atomic-clock levels. In most computers, the hardware clock is constructed from a high-frequency counter. In some computers, the value of this counter can be read from a device register, in which case the counter can be considered a high-resolution clock. Although this clock does not generate interrupts, it offers accurate measurements of time intervals.

### 12.3.4 Nonblocking and Asynchronous I/O

Another aspect of the system-call interface relates to the choice between blocking I/O and nonblocking I/O. When an application issues a **blocking** system call, the execution of the calling thread is suspended. The thread is moved from the operating system's run queue to a wait queue. After the system call completes, the thread is moved back to the run queue, where it is eligible to resume execution. When it resumes execution, it will receive the values returned by the system call. The physical actions performed by I/O devices are generally asynchronous—they take a varying or unpredictable amount of time. Nevertheless, operating systems provide blocking system calls for the application interface, because blocking application code is easier to write than nonblocking application code.

Some user-level processes need **nonblocking** I/O. One example is a user interface that receives keyboard and mouse input while processing and displaying data on the screen. Another example is a video application that reads frames from a file on disk while simultaneously decompressing and displaying the output on the display.

One way an application writer can overlap execution with I/O is to write a multithreaded application. Some threads can perform blocking system calls, while others continue executing. Some operating systems provide nonblocking I/O system calls. A nonblocking call does not halt the execution of the thread for an extended time. Instead, it returns quickly, with a return value that indicates how many bytes were transferred.

An alternative to a nonblocking system call is an asynchronous system call. An asynchronous call returns immediately, without waiting for the I/O to complete. The thread continues to execute its code. The completion of the I/O at some future time is communicated to the thread, either through the setting of some variable in the address space of the thread or through the triggering of a signal or software interrupt or a call-back routine that is executed outside the linear control flow of the thread. The difference between nonblocking and asynchronous system calls is that a nonblocking `read()` returns immediately with whatever data are available—the full number of bytes requested, fewer, or none at all. An asynchronous `read()` call requests a transfer that will be performed in its entirety but will complete at some future time. These two I/O methods are shown in Figure 12.9.

Asynchronous activities occur throughout modern operating systems. Frequently, they are not exposed to users or applications but rather are contained within the operating-system operation. Secondary storage device and network I/O are useful examples. By default, when an application issues a network send request or a storage device write request, the operating system notes the request, buffers the I/O, and returns to the application. When possible, to optimize overall system performance, the operating system completes the request. If a system failure occurs in the interim, the application will lose any "in-flight" requests. Therefore, operating systems usually put a limit on how

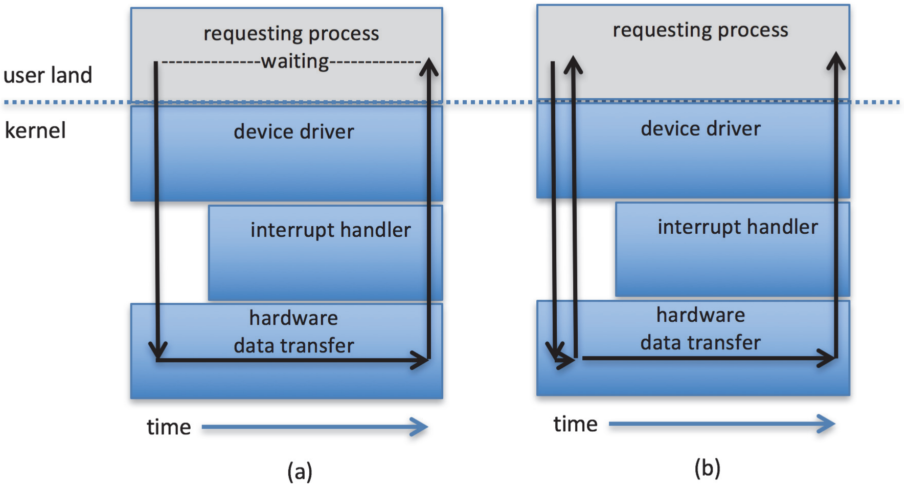


**Figure 12.9** Two I/O methods: (a) synchronous and (b) asynchronous.

long they will buffer a request. Some versions of UNIX flush their secondary storage buffers every 30 seconds, for example, or each request is flushed within 30 seconds of its occurrence. Systems provide a way to allow applications to request a flush of some buffers (like secondary storage buffers) so the data can be forced to secondary storage without waiting for the buffer flush interval. Data consistency within applications is maintained by the kernel, which reads data from its buffers before issuing I/O requests to devices, ensuring that data not yet written are nevertheless returned to a requesting reader. Note that multiple threads performing I/O to the same file might not receive consistent data, depending on how the kernel implements its I/O. In this situation, the threads may need to use locking protocols. Some I/O requests need to be performed immediately, so I/O system calls usually have a way to indicate that a given request, or I/O to a specific device, should be performed synchronously.

A good example of nonblocking behavior is the `select()` system call for network sockets. This system call takes an argument that specifies a maximum waiting time. By setting it to 0, a thread can poll for network activity without blocking. But using `select()` introduces extra overhead, because the `select()` call only checks whether I/O is possible. For a data transfer, `select()` must be followed by some kind of `read()` or `write()` command. A variation on this approach, found in Mach, is a blocking multiple-read call. It specifies desired reads for several devices in one system call and returns as soon as any one of them completes.

### 12.3.5 Vectored I/O

Some operating systems provide another major variation of I/O via their application interfaces. **Vectored I/O** allows one system call to perform multiple I/O operations involving multiple locations. For example, the UNIX `readv` system call accepts a vector of multiple buffers and either reads from a source to that vector or writes from that vector to a destination. The same transfer could be caused by several individual invocations of system calls, but this **scatter–gather** method is useful for a variety of reasons.

Multiple separate buffers can have their contents transferred via one system call, avoiding context-switching and system-call overhead. Without vectored I/O, the data might first need to be transferred to a larger buffer in the right order and then transmitted, which is inefficient. In addition, some versions of scatter–gather provide atomicity, assuring that all the I/O is done without interruption (and avoiding corruption of data if other threads are also performing I/O involving those buffers). When possible, programmers make use of scatter–gather I/O features to increase throughput and decrease system overhead.

## 12.4 Kernel I/O Subsystem

Kernels provide many services related to I/O. Several services—scheduling, buffering, caching, spooling, device reservation, and error handling—are provided by the kernel's I/O subsystem and build on the hardware and device-driver infrastructure. The I/O subsystem is also responsible for protecting itself from errant processes and malicious users.

### 12.4.1 I/O Scheduling

To schedule a set of I/O requests means to determine a good order in which to execute them. The order in which applications issue system calls rarely is the best choice. Scheduling can improve overall system performance, can share device access fairly among processes, and can reduce the average waiting time for I/O to complete. Here is a simple example to illustrate. Suppose that a disk arm is near the beginning of a disk and that three applications issue blocking read calls to that disk. Application 1 requests a block near the end of the disk, application 2 requests one near the beginning, and application 3 requests one in the middle of the disk. The operating system can reduce the distance that the disk arm travels by serving the applications in the order 2, 3, 1. Rearranging the order of service in this way is the essence of I/O scheduling.

Operating-system developers implement scheduling by maintaining a wait queue of requests for each device. When an application issues a blocking I/O system call, the request is placed on the queue for that device. The I/O scheduler rearranges the order of the queue to improve the overall system efficiency and the average response time experienced by applications. The operating system may also try to be fair, so that no one application receives especially poor service, or it may give priority service for delay-sensitive requests. For instance, requests from the virtual memory subsystem may take priority over application requests. Several scheduling algorithms for disk I/O were detailed in Section 11.2.

When a kernel supports asynchronous I/O, it must be able to keep track of many I/O requests at the same time. For this purpose, the operating system might attach the wait queue to a **device-status table**. The kernel manages this table, which contains an entry for each I/O device, as shown in Figure 12.10. Each table entry indicates the device's type, address, and state (not functioning,

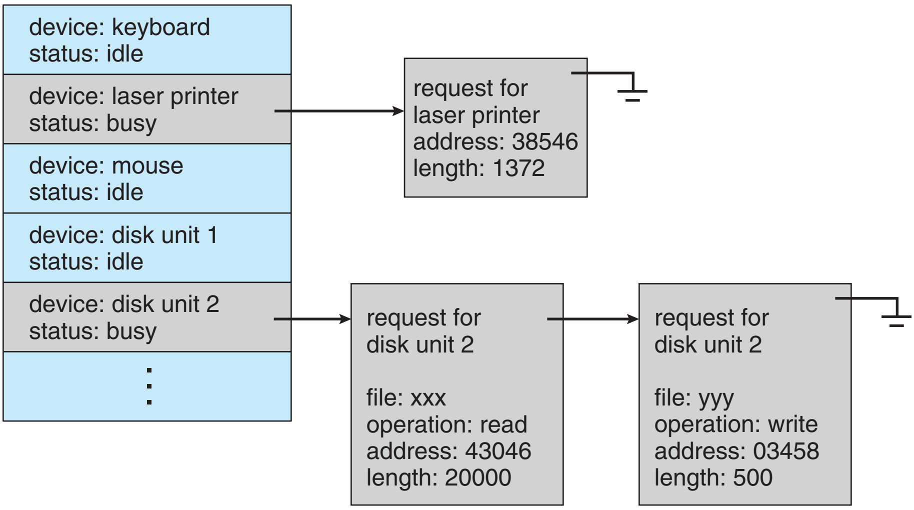

device: keyboard
status: idle
device: laser printer
status: busy
device: mouse
status: idle
device: disk unit 1
status: idle
device: disk unit 2
status: busy
request for
laser printer
address: 38546
length: 1372
request for
disk unit 2
file: xxx
operation: read
address: 43046
length: 20000
request for
disk unit 2
file: yyy
operation: write
address: 03458
length: 500

**Figure 12.10** Device-status table.

idle, or busy). If the device is busy with a request, the type of request and other parameters will be stored in the table entry for that device.

Scheduling I/O operations is one way in which the I/O subsystem improves the efficiency of the computer. Another way is by using storage space in main memory or elsewhere in the storage hierarchy via buffering, caching, and spooling.

### 12.4.2 Buffering

A **buffer**, of course, is a memory area that stores data being transferred between two devices or between a device and an application. Buffering is done for three reasons. One reason is to cope with a speed mismatch between the producer and consumer of a data stream. Suppose, for example, that a file is being received via Internet for storage on an SSD. The network speed may be a thousand times slower than the drive. So a buffer is created in main memory to accumulate the bytes received from the network. When an entire buffer of data has arrived, the buffer can be written to the drive in a single operation. Since the drive write is not instantaneous and the network interface still needs a place to store additional incoming data, two buffers are used. After the network fills the first buffer, the drive write is requested. The network then starts to fill the second buffer while the first buffer is written to storage. By the time the network has filled the second buffer, the drive write from the first one should have completed, so the network can switch back to the first buffer while the drive writes the second one. This **double buffering** decouples the producer of data from the consumer, thus relaxing timing requirements between them. The need for this decoupling is illustrated in Figure 12.11, which lists the enormous differences in device speeds for typical computer hardware and interfaces.

A second use of buffering is to provide adaptations for devices that have different data-transfer sizes. Such disparities are especially common in computer networking, where buffers are used widely for fragmentation and

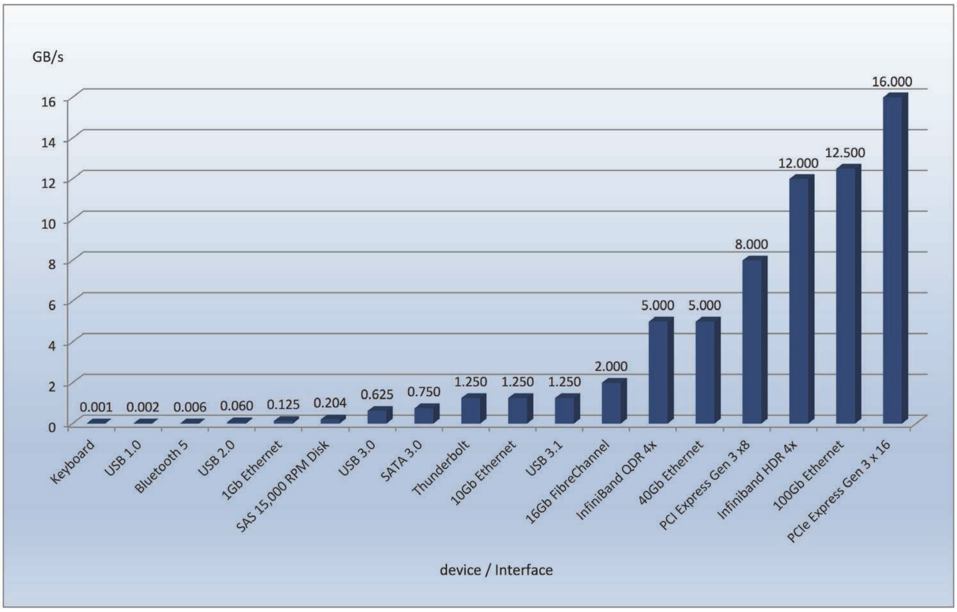


**Figure 12.11** Common PC and data-center I/O device and interface speeds.

reassembly of messages. At the sending side, a large message is fragmented into small network packets. The packets are sent over the network, and the receiving side places them in a reassembly buffer to form an image of the source data.

A third use of buffering is to support copy semantics for application I/O. An example will clarify the meaning of "copy semantics." Suppose that an application has a buffer of data that it wishes to write to disk. It calls the `write()` system call, providing a pointer to the buffer and an integer specifying the number of bytes to write. After the system call returns, what happens if the application changes the contents of the buffer? With **copy semantics**, the version of the data written to disk is guaranteed to be the version at the time of the application system call, independent of any subsequent changes in the application's buffer. A simple way in which the operating system can guarantee copy semantics is for the `write()` system call to copy the application data into a kernel buffer before returning control to the application. The disk write is performed from the kernel buffer, so that subsequent changes to the application buffer have no effect. Copying of data between kernel buffers and application data space is common in operating systems, despite the overhead that this operation introduces, because of the clean semantics. The same effect can be obtained more efficiently by clever use of virtual memory mapping and copy-on-write page protection.

### 12.4.3 Caching

A **cache** is a region of fast memory that holds copies of data. Access to the cached copy is more efficient than access to the original. For instance, the instructions of the currently running process are stored on disk, cached in physical memory, and copied again in the CPU's secondary and primary caches.

The difference between a buffer and a cache is that a buffer may hold the only existing copy of a data item, whereas a cache, by definition, holds a copy on faster storage of an item that resides elsewhere.

Caching and buffering are distinct functions, but sometimes a region of memory can be used for both purposes. For instance, to preserve copy semantics and to enable efficient scheduling of disk I/O, the operating system uses buffers in main memory to hold disk data. These buffers are also used as a cache, to improve the I/O efficiency for files that are shared by applications or that are being written and reread rapidly. When the kernel receives a file I/O request, the kernel first accesses the buffer cache to see whether that region of the file is already available in main memory. If it is, a physical disk I/O can be avoided or deferred. Also, disk writes are accumulated in the buffer cache for several seconds, so that large transfers are gathered to allow efficient write schedules. This strategy of delaying writes to improve I/O efficiency is discussed, in the context of remote file access, in Section 19.8.

### 12.4.4 Spooling and Device Reservation

A **spool** is a buffer that holds output for a device, such as a printer, that cannot accept interleaved data streams. Although a printer can serve only one job at a time, several applications may wish to print their output concurrently, without having their output mixed together. The operating system solves this problem by intercepting all output to the printer. Each application's output is spooled to a separate secondary storage file. When an application finishes printing, the spooling system queues the corresponding spool file for output to the printer. The spooling system copies the queued spool files to the printer one at a time. In some operating systems, spooling is managed by a system daemon process. In others, it is handled by an in-kernel thread. In either case, the operating system provides a control interface that enables users and system administrators to display the queue, remove unwanted jobs before those jobs print, suspend printing while the printer is serviced, and so on.

Some devices, such as tape drives and printers, cannot usefully multiplex the I/O requests of multiple concurrent applications. Spooling is one way operating systems can coordinate concurrent output. Another way to deal with concurrent device access is to provide explicit facilities for coordination. Some operating systems (including VMS) provide support for exclusive device access by enabling a process to allocate an idle device and to deallocate that device when it is no longer needed. Other operating systems enforce a limit of one open file handle to such a device. Many operating systems provide functions that enable processes to coordinate exclusive access among themselves. For instance, Windows provides system calls to wait until a device object becomes available. It also has a parameter to the `OpenFile()` system call that declares the types of access to be permitted to other concurrent threads. On these systems, it is up to the applications to avoid deadlock.

### 12.4.5 Error Handling

An operating system that uses protected memory can guard against many kinds of hardware and application errors, so that a complete system failure is not the usual result of each minor mechanical malfunction. Devices and I/O transfers can fail in many ways, either for transient reasons, as when a network becomes overloaded, or for “permanent” reasons, as when a disk controller becomes defective. Operating systems can often compensate effectively for transient failures. For instance, a disk `read()` failure results in a `read()` retry, and a network `send()` error results in a `resend()`, if the protocol so specifies. Unfortunately, if an important component experiences a permanent failure, the operating system is unlikely to recover.

As a general rule, an I/O system call will return one bit of information about the status of the call, signifying either success or failure. In the UNIX operating system, an additional integer variable named `errno` is used to return an error code—one of about a hundred values—indicating the general nature of the failure (for example, argument out of range, bad pointer, or file not open). By contrast, some hardware can provide highly detailed error information, although many current operating systems are not designed to convey this information to the application. For instance, a failure of a SCSI device is reported by the SCSI protocol in three levels of detail: a **sense key** that identifies the general nature of the failure, such as a hardware error or an illegal request; an **additional sense code** that states the category of failure, such as a bad command parameter or a self-test failure; and an **additional sense-code qualifier** that gives even more detail, such as which command parameter was in error or which hardware subsystem failed its self-test. Further, many SCSI devices maintain internal pages of error-log information that can be requested by the host—but seldom are.

### 12.4.6 I/O Protection

Errors are closely related to the issue of protection. A user process may accidentally or purposely attempt to disrupt the normal operation of a system by attempting to issue illegal I/O instructions. We can use various mechanisms to ensure that such disruptions cannot take place in the system.

To prevent users from performing illegal I/O, we define all I/O instructions to be privileged instructions. Thus, users cannot issue I/O instructions directly; they must do it through the operating system. To do I/O, a user program executes a system call to request that the operating system perform I/O on its behalf (Figure 12.12). The operating system, executing in monitor mode, checks that the request is valid and, if it is, does the I/O requested. The operating system then returns to the user.

In addition, any memory-mapped and I/O port memory locations must be protected from user access by the memory-protection system. Note that a kernel cannot simply deny all user access. Most graphics games and video editing and playback software need direct access to memory-mapped graphics controller memory to speed the performance of the graphics, for example. The kernel might in this case provide a locking mechanism to allow a section of graphics memory (representing a window on screen) to be allocated to one process at a time.

### 12.4.7 Kernel Data Structures

The kernel needs to keep state information about the use of I/O components. It does so through a variety of in-kernel data structures, such as the open-file table

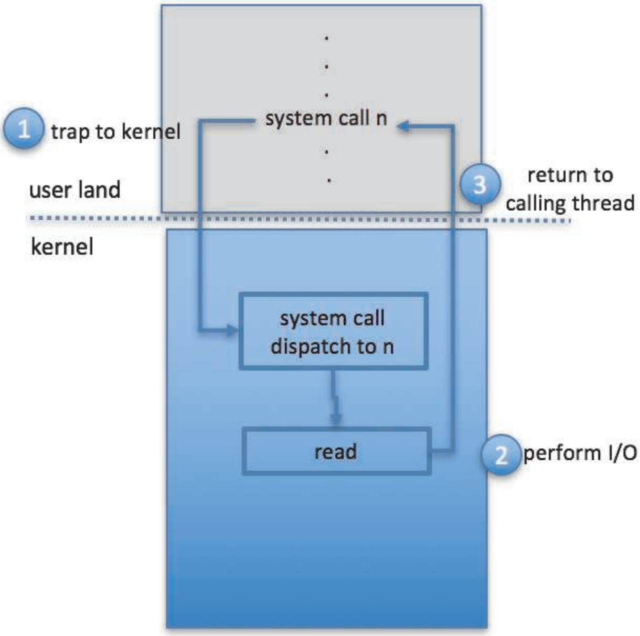


Figure 12.12 Use of a system call to perform I/O.

structure discussed in Section 14.1. The kernel uses many similar structures to track network connections, character-device communications, and other I/O activities.

UNIX provides file-system access to a variety of entities, such as user files, raw devices, and the address spaces of processes. Although each of these entities supports a `read()` operation, the semantics differ. For instance, to read a user file, the kernel needs to probe the buffer cache before deciding whether to perform a disk I/O. To read a raw disk, the kernel needs to ensure that the request size is a multiple of the disk sector size and is aligned on a sector boundary. To read a process image, it is merely necessary to copy data from memory. UNIX encapsulates these differences within a uniform structure by using an object-oriented technique. The open-file record, shown in Figure 12.13, contains a dispatch table that holds pointers to the appropriate routines, depending on the type of file.

Some operating systems use object-oriented methods even more extensively. For instance, Windows uses a message-passing implementation for I/O. An I/O request is converted into a message that is sent through the kernel to the I/O manager and then to the device driver, each of which may change the message contents. For output, the message contains the data to be written. For input, the message contains a buffer to receive the data. The message-passing approach can add overhead, by comparison with procedural techniques that use shared data structures, but it simplifies the structure and design of the I/O system and adds flexibility.

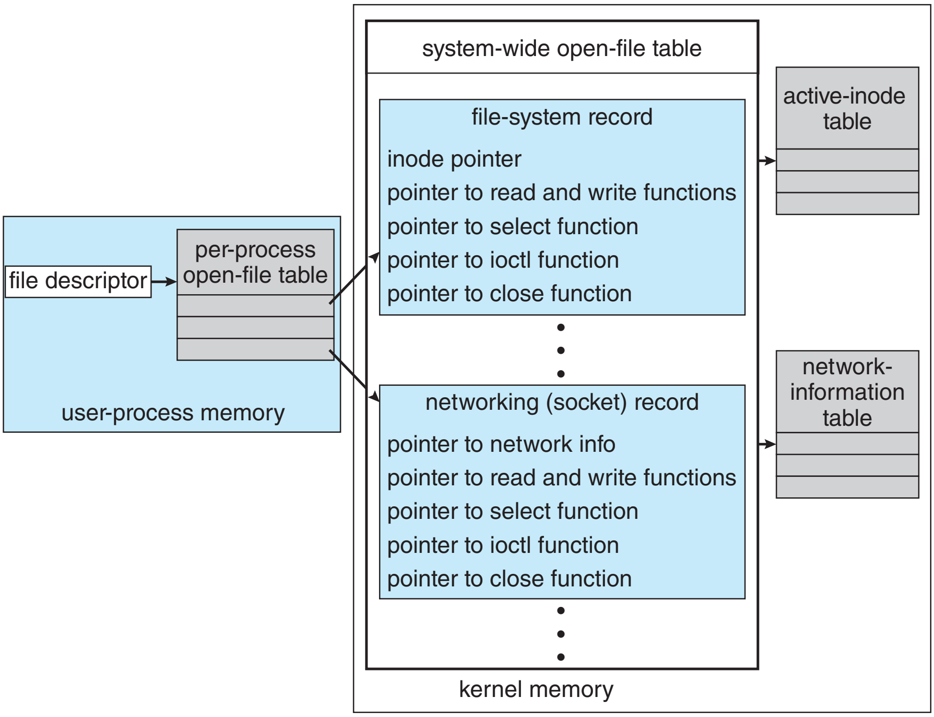

system-wide open-file table
file-system record
inode pointer
pointer to read and write functions
pointer to select function
pointer to ioctl function
pointer to close function
networking (socket) record
pointer to network info
pointer to read and write functions
pointer to select function
pointer to ioctl function
pointer to close function
kernel memory
active-inode table
network-information table
file descriptor
per-process open-file table
user-process memory

**Figure 12.13** UNIX I/O kernel structure.

### 12.4.8 Power Management

Computers residing in data centers may seem far removed from issues of power use, but as power costs increase and the world becomes increasingly troubled about the long-term effects of greenhouse gas emissions, data centers have become a cause for concern and a target for increased efficiencies. Electricity use generates heat, and computer components can fail due to high temperatures, so cooling is part of the equation as well. Consider that cooling a modern data center may use twice as much electricity as powering the equipment does. Many approaches to data-center power optimization are in use, ranging from interchanging data-center air without side air, chilling with natural sources such as lake water, and solar panels.

Operating systems play a role in power use (and therefore heat generation and cooling). In cloud computing environments, processing loads can be adjusted by monitoring and management tools to evacuate all user processes from systems, idling those systems and powering them off until the load requires their use. An operating system could analyze its load and, if sufficiently low and hardware-enabled, power off components such as CPUsand external I/O devices.

CPU cores can be suspended when the system load does not require them and resumed when the load increases and more cores are needed to run the queue of threads. Their state, of course, needs to be saved on suspend and restored on resume. This feature is needed in servers because a data center full of servers can use vast amounts of electricity, and disabling unneeded cores can decrease electricity (and cooling) needs.

In mobile computing, power management becomes a high-priority aspect of the operating system. Minimizing power use and therefore maximizing battery life increases the usability of a device and helps it compete with alternative devices. Today's mobile devices offer the functionality of yesterday's high-end desktop, yet are powered by batteries and are small enough to fit in your pocket. In order to provide satisfactory battery life, modern mobile operating systems are designed from the ground up with power management as a key feature. Let's examine in detail three major features that enable the popular Android mobile system to maximize battery life: power collapse, component-level power management, and wakelocks.

Power collapse is the ability to put a device into a very deep sleep state. The device uses only marginally more power than if it were fully powered off, yet it is still able to respond to external stimuli, such as the user pressing a button, at which time it quickly powers back on. Power collapse is achieved by powering off many of the individual components within a device—such as the screen, speakers, and I/O subsystem—so that they consume no power. The operating system then places the CPU in its lowest sleep state. A modern ARM CPU might consume hundreds of milliwatts per core under typical load yet only a handful of milliwatts in its lowest sleep state. In such a state, although the CPU is idle, it can receive an interrupt, wake up, and resume its previous activity very quickly. Thus, an idle Android phone in your pocket uses very little power, but it can spring to life when it receives a phone call.

How is Android able to turn off the individual components of a phone? How does it know when it is safe to power off the flash storage, and how does it know to do that before powering down the overall I/O subsystem? The answer is component-level power management, which is an infrastructure that understands the relationship between components and whether each component is in use. To understand the relationship between components, Android builds a device tree that represents the phone's physical-device topology. For example, in such a topology, flash and USB storage would be sub-nodes of the I/O subsystem, which is a sub-node of the system bus, which in turn connects to the CPU. To understand usage, each component is associated with its device driver, and the driver tracks whether the component is in use—for example, if there is I/O pending to flash or if an application has an open reference to the audio subsystem. With this information, Android can manage the power of the phone's individual components: If a component is unused, it is turned off. If all of the components on, say, the system bus are unused, the system bus is turned off. And if all of the components in the entire device tree are unused, the system may enter power collapse.

With these technologies, Android can aggressively manage its power consumption. But a final piece of the solution is missing: the ability for applications to temporarily prevent the system from entering power collapse. Consider a user playing a game, watching a video, or waiting for a web page to open. In all of these cases, the application needs a way to keep the device awake, at least temporarily. Wakelocks enable this functionality. Applications acquire and release wakelocks as needed. While an application holds a wakelock, the kernel will prevent the system from entering power collapse. For example, while the Android Market is updating an application, it will hold a wakelock to ensure that the system does not go to sleep until the update is complete. Once complete, the Android Market will release the wakelock, allowing the system to enter power collapse.

Power management in general is based on device management, which is more complicated than we have so far portrayed it. At boot time, the firmware system analyzes the system hardware and creates a device tree in RAM. The kernel then uses that device tree to load device drivers and manage devices. Many additional activities pertaining to devices must be managed, though, including addition and subtraction of devices from a running system ("hot-plug"), understanding and changing device states, and power management. Modern general-purpose computers use another set of firmware code, **advanced configuration and power interface** (**ACPI**), to manage these aspects of hardware. ACPI is an industry standard (http://www.acpi.info) with many features. It provides code that runs as routines callable by the kernel for device state discovery and management, device error management, and power management. For example, when the kernel needs to quiesce a device, it calls the device driver, which calls the ACPI routines, which then talk to the device.

### 12.4.9 Kernel I/O Subsystem Summary

In summary, the I/O subsystem coordinates an extensive collection of services that are available to applications and to other parts of the kernel. The I/O subsystem supervises these procedures:

- Management of the name space for files and devices
- Access control to files and devices
- Operation control (for example, a modem cannot `seek()`)
- File-system space allocation
- Device allocation
- Buffering, caching, and spooling
- I/O scheduling
- Device-status monitoring, error handling, and failure recovery
- Device-driver configuration and initialization
- Power management of I/O devices

The upper levels of the I/O subsystem access devices via the uniform interface provided by the device drivers.

## 12.5 Transforming I/O Requests to Hardware Operations

Earlier, we described the handshaking between a device driver and a device controller, but we did not explain how the operating system connects an application request to a set of network wires or to a specific disk sector. Consider, for example, reading a file from disk. The application refers to the data by a file name. Within a disk, the file system maps from the file name through the file-system directories to obtain the space allocation of the file. For instance, in MS-DOS for FAT (a relatively simple operating and file system still used today as a common interchange format), the name maps to a number that indicates an entry in the file-access table, and that table entry tells which disk blocks are allocated to the file. In UNIX, the name maps to an inode number, and the corresponding inode contains the space-allocation information. But how is the connection made from the file name to the disk controller (the hardware port address or the memory-mapped controller registers)?

One method is that used by MS-DOS for FAT, mentioned above. The first part of an MS-DOS file name, preceding the colon, is a string that identifies a specific hardware device. For example, `C:` is the first part of every file name on the primary hard disk. The fact that `C:` represents the primary hard disk is built into the operating system; `C:` is mapped to a specific port address through a device table. Because of the colon separator, the device name space is separate from the file-system name space. This separation makes it easy for the operating system to associate extra functionality with each device. For instance, it is easy to invoke spooling on any files written to the printer.

If, instead, the device name space is incorporated in the regular file-system name space, as it is in UNIX, the normal file-system name services are provided automatically. If the file system provides ownership and access control to all file names, then devices have owners and access control. Since files are stored on devices, such an interface provides access to the I/O system at two levels. Names can be used to access the devices themselves or to access the files stored on the devices.

UNIX represents device names in the regular file-system name space. Unlike an MS-DOS FAT file name, which has a colon separator, a UNIX path name has no clear separation of the device portion. In fact, no part of the path name is the name of a device. UNIX has a **mount table** that associates prefixes of path names with specific device names. To resolve a path name, UNIX looks up the name in the mount table to find the longest matching prefix; the corresponding entry in the mount table gives the device name. This device name also has the form of a name in the file-system name space. When UNIX looks up this name in the file-system directory structures, it finds not an inode number but a `<major, minor>` device number. The major device number identifies a device driver that should be called to handle I/O to this device. The minor device number is passed to the device driver to index into a device table. The corresponding device-table entry gives the port address or the memory-mapped address of the device controller.

Modern operating systems gain significant flexibility from the multiple stages of lookup tables in the path between a request and a physical device controller. The mechanisms that pass requests between applications and drivers are general. Thus, we can introduce new devices and drivers into a computer without recompiling the kernel. In fact, some operating systems have the ability to load device drivers on demand. At boot time, the system first probes the hardware buses to determine what devices are present. It then loads the necessary drivers, either immediately or when first required by an I/O request. Devices added after boot can be detected by the error they cause (interrupt-generated with no associated interrupt handler, for example), which can prompt the kernel to inspect the device details and load an appropriate device driver dynamically. Of course, dynamic loading (and unloading) is more complicated than static loading, requiring more complex kernel algorithms, device-structure locking, error handling, and so forth.

We next describe the typical life cycle of a blocking read request, as depicted in Figure 12.14. The figure suggests that an I/O operation requires a great many steps that together consume a tremendous number of CPU cycles.

1. A process issues a blocking `read()` system call to a file descriptor of a file that has been opened previously.
2. The system-call code in the kernel checks the parameters for correctness. In the case of input, if the data are already available in the buffer cache, the data are returned to the process, and the I/O request is completed.
3. Otherwise, a physical I/O must be performed. The process is removed from the run queue and is placed on the wait queue for the device, and the I/O request is scheduled. Eventually, the I/O subsystem sends the request to the device driver. Depending on the operating system, the request is sent via a subroutine call or an in-kernel message.

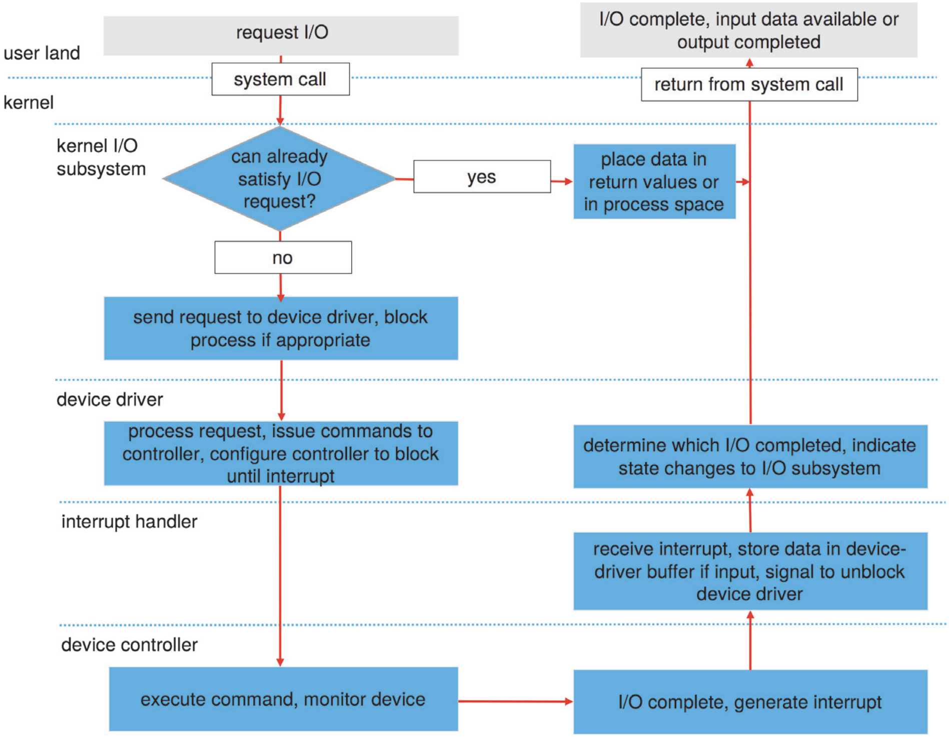

user land
request I/O
I/O complete, input data available or output completed
system call
return from system call
kernel
kernel I/O subsystem
can already satisfy I/O request?
yes
place data in return values or in process space
no
send request to device driver, block process if appropriate
device driver
process request, issue commands to controller, configure controller to block until interrupt
determine which I/O completed, indicate state changes to I/O subsystem
interrupt handler
receive interrupt, store data in device-driver buffer if input, signal to unblock device driver
device controller
execute command, monitor device
I/O complete, generate interrupt

**Figure 12.14** The life cycle of an I/O request.

4. The device driver allocates kernel buffer space to receive the data and schedules the I/O. Eventually, the driver sends commands to the device controller by writing into the device-control registers.
5. The device controller operates the device hardware to perform the data transfer.
6. The driver may poll for status and data, or it may have set up a DMA transfer into kernel memory. We assume that the transfer is managed by a DMA controller, which generates an interrupt when the transfer completes.
7. The correct interrupt handler receives the interrupt via the interrupt-vector table, stores any necessary data, signals the device driver, and returns from the interrupt.
8. The device driver receives the signal, determines which I/O request has completed, determines the request's status, and signals the kernel I/O subsystem that the request has been completed.
9. The kernel transfers data or return codes to the address space of the requesting process and moves the process from the wait queue back to the ready queue.
10. Moving the process to the ready queue unblocks the process. When the scheduler assigns the process to the CPU, the process resumes execution at the completion of the system call.

## 12.6 STREAMS

UNIX System V (and many subsequent UNIX releases) has an interesting mechanism, called **STREAMS**, that enables an application to assemble pipelines of driver code dynamically. A stream is a full-duplex connection between a device driver and a user-level process. It consists of a **stream head** that interfaces with the user process, a **driver end** that controls the device, and zero or more **stream modules** between the stream head and the driver end. Each of these components contains a pair of queues—a read queue and a write queue. Message passing is used to transfer data between queues. The STREAMS structure is shown in Figure 12.15.

Modules provide the functionality of STREAMS processing; they are ***pushed*** onto a stream by use of the `ioctl()` system call. For example, a process can open a USB device (like a keyboard) via a stream and can push on a module to handle input editing. Because messages are exchanged between queues in adjacent modules, a queue in one module may overflow an adjacent queue. To prevent this from occurring, a queue may support **flow control**. Without flow control, a queue accepts all messages and immediately sends them on to the queue in the adjacent module without buffering them. A queue that supports flow control buffers messages and does not accept messages without sufficient buffer space. This process involves exchanges of control messages between queues in adjacent modules.

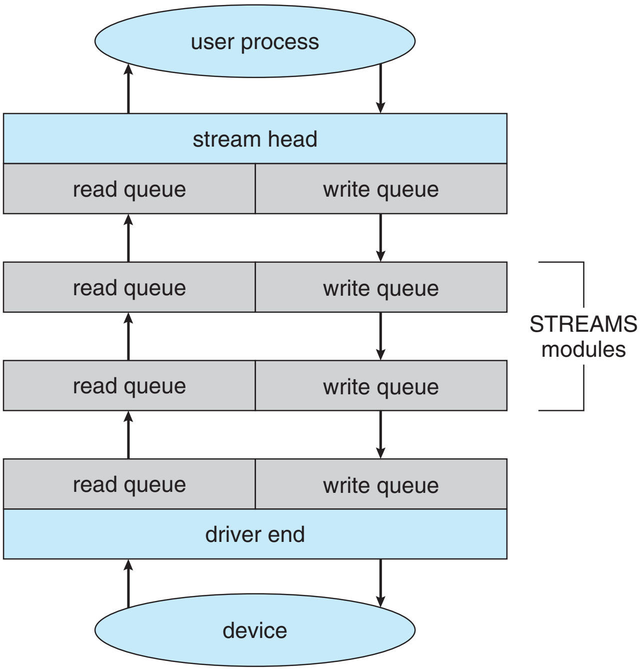


**Figure 12.15** The STREAMS structure.

A user process writes data to a device using either the `write()` or `putmsg()` system call. The `write()` system call writes raw data to the stream, whereas `putmsg()` allows the user process to specify a message. Regardless of the system call used by the user process, the stream head copies the data into a message and delivers it to the queue for the next module in line. This copying of messages continues until the message is copied to the driver end and hence the device. Similarly, the user process reads data from the stream head using either the `read()` or `getmsg()` system call. If `read()` is used, the stream head gets a message from its adjacent queue and returns ordinary data (an unstructured byte stream) to the process. If `getmsg()` is used, a message is returned to the process.

STREAMS I/O is asynchronous (or nonblocking) except when the user process communicates with the stream head. When writing to the stream, the user process will block, assuming the next queue uses flow control, until there is room to copy the message. Likewise, the user process will block when reading from the stream until data are available.

As mentioned, the driver end—like the stream head and modules—has a read and write queue. However, the driver end must respond to interrupts, such as one triggered when a frame is ready to be read from a network. Unlike the stream head, which may block if it is unable to copy a message to the next queue in line, the driver end must handle all incoming data. Drivers must support flow control as well. However, if a device's buffer is full, the device typically resorts to dropping incoming messages. Consider a network card whose input buffer is full. The network card must simply drop further messages until there is enough buffer space to store incoming messages.

The benefit of using STREAMS is that it provides a framework for a modular and incremental approach to writing device drivers and network protocols. Modules may be used by different streams and hence by different devices. For example, a networking module may be used by both an Ethernet network card and a 802.11 wireless network card. Furthermore, rather than treating character-device I/O as an unstructured byte stream, STREAMS allows support for message boundaries and control information when communicating between modules. Most UNIX variants support STREAMS, and it is the preferred method for writing protocols and device drivers. For example, System V UNIX and Solaris implement the socket mechanism using STREAMS.

## 12.7 Performance

I/O is a major factor in system performance. It places heavy demands on the CPU to execute device-driver code and to schedule processes fairly and efficiently as they block and unblock. The resulting context switches stress the CPU and its hardware caches. I/O also exposes any inefficiencies in the interrupt-handling mechanisms in the kernel. In addition, I/O loads down the memory bus during data copies between controllers and physical memory and again during copies between kernel buffers and application data space. Coping gracefully with all these demands is one of the major concerns of a computer architect.

Although modern computers can handle many thousands of interrupts per second, interrupt handling is a relatively expensive task. Each interrupt causes the system to perform a state change, to execute the interrupt handler, and then to restore state. Programmed I/O can be more efficient than interrupt-driven I/O, if the number of cycles spent in busy waiting is not excessive. An I/O completion typically unblocks a process, leading to the full overhead of a context switch.

Network traffic can also cause a high context-switch rate. Consider, for instance, a remote login from one machine to another. Each character typed on the local machine must be transported to the remote machine. On the local machine, the character is typed; a keyboard interrupt is generated; and the character is passed through the interrupt handler to the device driver, to the kernel, and then to the user process. The user process issues a network I/O system call to send the character to the remote machine. The character then flows into the local kernel, through the network layers that construct a network packet, and into the network device driver. The network device driver transfers the packet to the network controller, which sends the character and generates an interrupt. The interrupt is passed back up through the kernel to cause the network I/O system call to complete.

Now, the remote system's network hardware receives the packet, and an interrupt is generated. The character is unpacked from the network protocols and is given to the appropriate network daemon. The network daemon identifies which remote login session is involved and passes the packet to the appropriate subdaemon for that session. Throughout this flow, there are context switches and state switches (Figure 12.16). Usually, the receiver echoes the character back to the sender; that approach doubles the work.

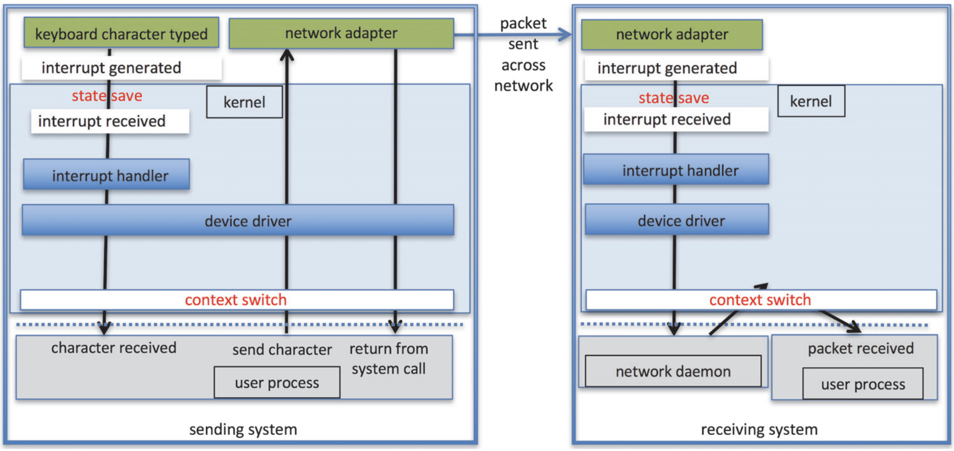

keyboard character typed
network adapter
interrupt generated
state save
kernel
interrupt received
interrupt handler
device driver
context switch
character received
send character
return from system call
user process
sending system
packet sent across network
network adapter
interrupt generated
state save
kernel
interrupt received
interrupt handler
device driver
context switch
network daemon
packet received
user process
receiving system

**Figure 12.16** Intercomputer communications.

Some systems use separate **front-end processors** for terminal I/O to reduce the interrupt burden on the main CPU. For instance, a **terminal concentrator** can multiplex the traffic from hundreds of remote terminals into one port on a large computer. An **I/O channel** is a dedicated, special-purpose CPU found in mainframes and in other high-end systems. The job of a channel is to offload I/O work from the main CPU. The idea is that the channels keep the data flowing smoothly, while the main CPU remains free to process the data. Like the device controllers and DMA controllers found in smaller computers, a channel can process more general and sophisticated programs, so channels can be tuned for particular workloads.

We can employ several principles to improve the efficiency of I/O:

- Reduce the number of context switches.
- Reduce the number of times that data must be copied in memory while passing between device and application.
- Reduce the frequency of interrupts by using large transfers, smart controllers, and polling (if busy waiting can be minimized).
- Increase concurrency by using DMA-knowledgeable controllers or channels to offload simple data copying from the CPU.
- Move processing primitives into hardware, to allow their operation in device controllers to be concurrent with CPU and bus operation.
- Balance CPU, memory subsystem, bus, and I/O performance, because an overload in any one area will cause idleness in others.

I/O devices vary greatly in complexity. For instance, a mouse is simple. The mouse movements and button clicks are converted into numeric values that are passed from hardware, through the mouse device driver, to the application. By contrast, the functionality provided by the Windows disk device driver is complex. It not only manages individual disks but also implements RAID arrays

(Section 11.8). To do so, it converts an application's read or write request into a coordinated set of disk I/O operations. Moreover, it implements sophisticated error-handling and data-recovery algorithms and takes many steps to optimize disk performance.

Where should the I/O functionality be implemented—in the device hardware, in the device driver, or in application software? Sometimes we observe the progression depicted in Figure 12.17.

- Initially, we implement experimental I/O algorithms at the application level, because application code is flexible and application bugs are unlikely to cause system crashes. Furthermore, by developing code at the application level, we avoid the need to reboot or reload device drivers after every change to the code. An application-level implementation can be inefficient, however, because of the overhead of context switches and because the application cannot take advantage of internal kernel data structures and kernel functionality (such as efficient in-kernel messaging, threading, and locking). The FUSE system interface, for example, allows file systems to be written and run in user mode.
- When an application-level algorithm has demonstrated its worth, we may reimplement it in the kernel. This can improve performance, but the development effort is more challenging, because an operating-system kernel is a large, complex software system. Moreover, an in-kernel implementation must be thoroughly debugged to avoid data corruption and system crashes.
- The highest performance may be obtained through a specialized implementation in hardware, either in the device or in the controller. The disadvantages of a hardware implementation include the difficulty and expense of making further improvements or of fixing bugs, the increased devel-

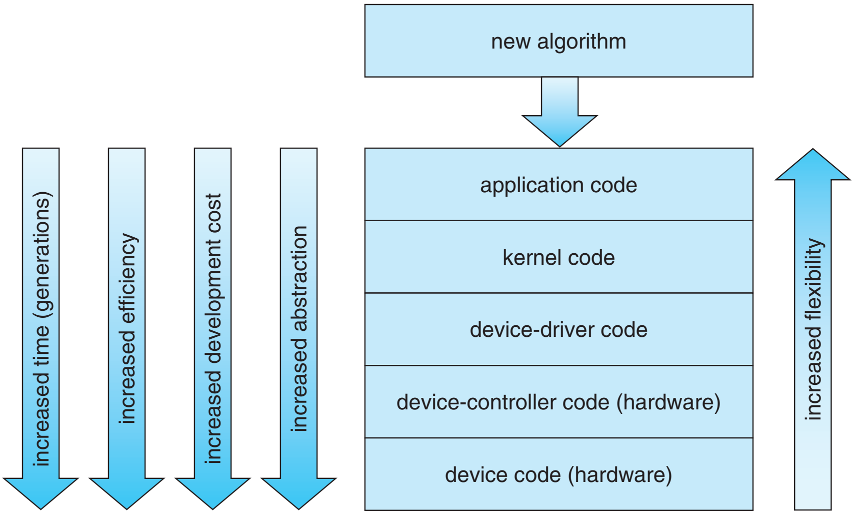

new algorithm
application code
kernel code
device-driver code
device-controller code (hardware)
device code (hardware)
increased time (generations)
increased efficiency
increased development cost
increased abstraction
increased flexibility

**Figure 12.17** Device functionality progression.

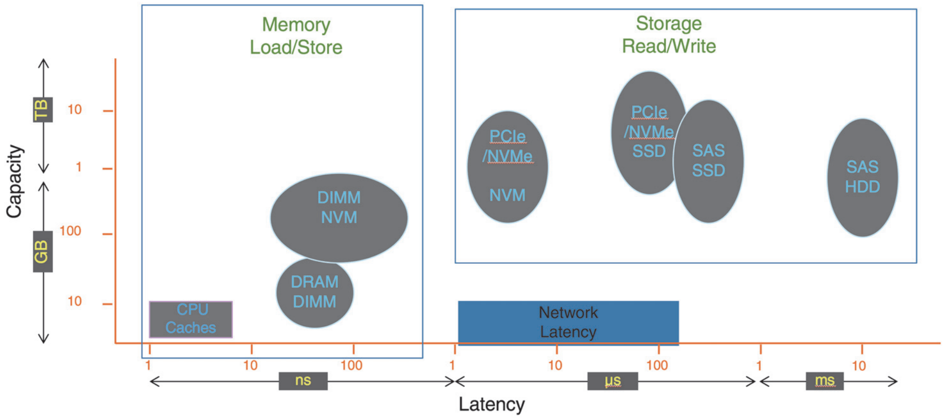


**Figure 12.18** I/O performance of storage (and network latency).

opment time (months rather than days), and the decreased flexibility. For instance, a hardware RAID controller may not provide any means for the kernel to influence the order or location of individual block reads and writes, even if the kernel has special information about the workload that would enable it to improve the I/O performance.

Over time, as with other aspects of computing, I/O devices have been increasing in speed. Nonvolatile memory devices are growing in popularity and in the variety of devices available. The speed of NVM devices varies from high to extraordinary, with next-generation devices nearing the speed of DRAM. These developments are increasing pressure on I/O subsystems as well as operating system algorithms to take advantage of the read/write speeds now available. Figure 12.18 shows CPU and storage devices in two dimensions: capacity and latency of I/O operations. Added to the figure is a representation of networking latency to reveal the performance "tax" networking adds to I/O.

## 12.8 Summary

- The basic hardware elements involved in I/O are buses, device controllers, and the devices themselves.
- The work of moving data between devices and main memory is performed by the CPU as programmed I/O or is offloaded to a DMA controller.
- The kernel module that controls a device is a device driver. The system-call interface provided to applications is designed to handle several basic categories of hardware, including block devices, character-stream devices, memory-mapped files, network sockets, and programmed interval timers. The system calls usually block the processes that issue them, but nonblocking and asynchronous calls are used by the kernel itself and by applications that must not sleep while waiting for an I/O operation to complete.

- The kernel's I/O subsystem provides numerous services. Among these are I/O scheduling, buffering, caching, spooling, device reservation, error handling. Another service, name translation, makes the connections between hardware devices and the symbolic file names used by applications. It involves several levels of mapping that translate from character-string names, to specific device drivers and device addresses, and then to physical addresses of I/O ports or bus controllers. This mapping may occur within the file-system name space, as it does in UNIX, or in a separate device name space, as it does in MS-DOS.
- STREAMS is an implementation and methodology that provides a framework for a modular and incremental approach to writing device drivers and network protocols. Through STREAMS, drivers can be stacked, with data passing through them sequentially and bidirectionally for processing.
- I/O system calls are costly in terms of CPU consumption because of the many layers of software between a physical device and an application. These layers imply overhead from several sources: context switching to cross the kernel's protection boundary, signal and interrupt handling to service the I/O devices, and the load on the CPU and memory system to copy data between kernel buffers and application space.

## Practice Exercises

**12.1** State three advantages of placing functionality in a device controller, rather than in the kernel. State three disadvantages.

**12.2** The example of handshaking in Section 12.2 used two bits: a busy bit and a command-ready bit. Is it possible to implement this handshaking with only one bit? If it is, describe the protocol. If it is not, explain why one bit is insufficient.

**12.3** Why might a system use interrupt-driven I/O to manage a single serial port and polling I/O to manage a front-end processor, such as a terminal concentrator?

**12.4** Polling for an I/O completion can waste a large number of CPU cycles if the processor iterates a busy-waiting loop many times before the I/O completes. But if the I/O device is ready for service, polling can be much more efficient than catching and dispatching an interrupt. Describe a hybrid strategy that combines polling, sleeping, and interrupts for I/O device service. For each of these three strategies (pure polling, pure interrupts, hybrid), describe a computing environment in which that strategy is more efficient than either of the others.

**12.5** How does DMA increase system concurrency? How does it complicate hardware design?

**12.6** Why is it important to scale up system-bus and device speeds as CPU speed increases?

**12.7** Distinguish between a driver end and a stream module in a STREAMS operation.

## Further Reading

[Hennessy and Patterson (2012)] describe multiprocessor systems and cache-consistency issues. [Intel (2011)] is a good source of information for Intel processors.

Details about PCIe can be found at https://pcisig.com. For more about ACPI see http://www.acpi.info.

The use of FUSE for user-mode file systems can create performance problems. An analysis of those issues can be found in https://www.usenix.org/conference/fast17/technical-sessions/presentation/vangoor.

## Bibliography

**[Hennessy and Patterson (2012)]** J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Fifth Edition, Morgan Kaufmann (2012).

**[Intel (2011)]** *Intel 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 3A and 3B*. Intel Corporation (2011).

## Chapter 12 Exercises

**12.8** When multiple interrupts from different devices appear at about the same time, a priority scheme could be used to determine the order in which the interrupts would be serviced. Discuss what issues need to be considered in assigning priorities to different interrupts.

**12.9** What are the advantages and disadvantages of supporting memory-mapped I/O to device-control registers?

**12.10** Consider the following I/O scenarios on a single-user PC:

a. A mouse used with a graphical user interface

b. A tape drive on a multitasking operating system (with no device preallocation available)

c. A disk drive containing user files

d. A graphics card with direct bus connection, accessible through memory-mapped I/O

For each of these scenarios, would you design the operating system to use buffering, spooling, caching, or a combination? Would you use polled I/O or interrupt-driven I/O? Give reasons for your choices.

**12.11** In most multiprogrammed systems, user programs access memory through virtual addresses, while the operating system uses raw physical addresses to access memory. What are the implications of this design for the initiation of I/O operations by the user program and their execution by the operating system?

**12.12** What are the various kinds of performance overhead associated with servicing an interrupt?

**12.13** Describe three circumstances under which blocking I/O should be used. Describe three circumstances under which nonblocking I/O should be used. Why not just implement nonblocking I/O and have processes busy-wait until their devices are ready?

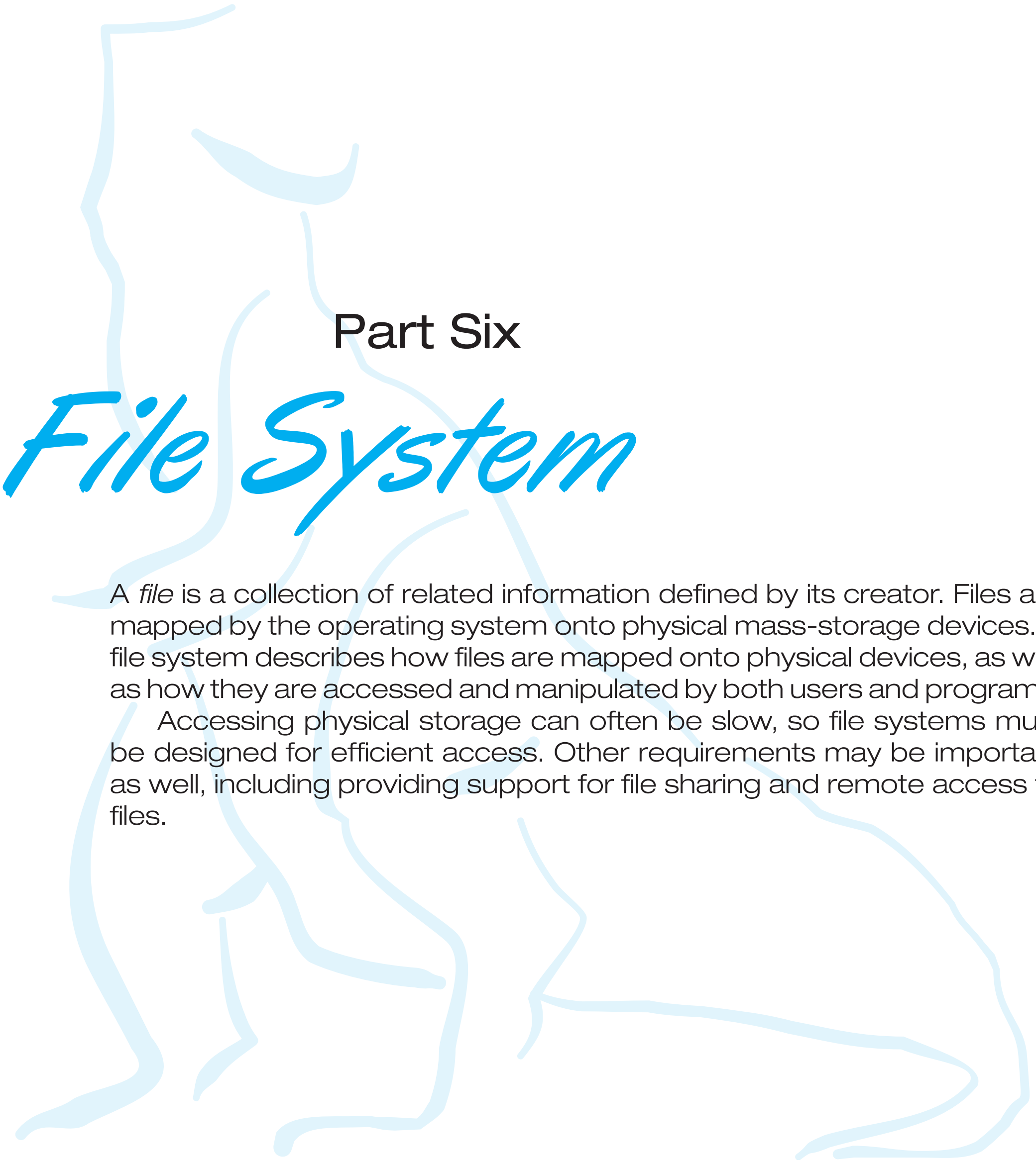
**12.14** Typically, at the completion of a device I/O, a single interrupt is raised and appropriately handled by the host processor. In certain settings, however, the code that is to be executed at the completion of the I/O can be broken into two separate pieces. The first piece executes immediately after the I/O completes and schedules a second interrupt for the remaining piece of code to be executed at a later time. What is the purpose of using this strategy in the design of interrupt handlers?

**12.15** Some DMA controllers support direct virtual memory access, where the targets of I/O operations are specified as virtual addresses and a translation from virtual to physical address is performed during the DMA. How does this design complicate the design of the DMA controller? What are the advantages of providing such functionality?

**12.16** UNIX coordinates the activities of the kernel I/O components by manipulating shared in-kernel data structures, whereas Windows uses object-oriented message passing between kernel I/O components. Discuss three pros and three cons of each approach.

**12.17** Write (in pseudocode) an implementation of virtual clocks, including the queueing and management of timer requests for the kernel and applications. Assume that the hardware provides three timer channels.

**12.18** Discuss the advantages and disadvantages of guaranteeing reliable transfer of data between modules in the STREAMS abstraction.



# Part Six

# File System

A *file* is a collection of related information defined by its creator. Files are mapped by the operating system onto physical mass-storage devices. A file system describes how files are mapped onto physical devices, as well as how they are accessed and manipulated by both users and programs.

Accessing physical storage can often be slow, so file systems must be designed for efficient access. Other requirements may be important as well, including providing support for file sharing and remote access to files.

CHAPTER 13

# File-System Interface

For most users, the file system is the most visible aspect of a general-purpose operating system. It provides the mechanism for on-line storage of and access to both data and programs of the operating system and all the users of the computer system. The file system consists of two distinct parts: a collection of files, each storing related data, and a directory structure, which organizes and provides information about all the files in the system. Most file systems live on storage devices, which we described in Chapter 11 and will continue to discuss in the next chapter. In this chapter, we consider the various aspects of files and the major directory structures. We also discuss the semantics of sharing files among multiple processes, users, and computers. Finally, we discuss ways to handle file protection, necessary when we have multiple users and want to control who may access files and how files may be accessed.

## CHAPTER OBJECTIVES

- Explain the function of file systems.
- Describe the interfaces to file systems.
- Discuss file-system design tradeoffs, including access methods, file sharing, file locking, and directory structures.
- Explore file-system protection.

## 13.1 File Concept

Computers can store information on various storage media, such as NVM devices, HDDs, magnetic tapes, and optical disks. So that the computer system will be convenient to use, the operating system provides a uniform logical view of stored information. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the **file**. Files are mapped by the operating system onto physical devices. These storage devices are usually nonvolatile, so the contents are persistent between system reboots.

A file is a named collection of related information that is recorded on secondary storage. From a user's perspective, a file is the smallest allotment of logical secondary storage; that is, data cannot be written to secondary storage unless they are within a file. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be free form, such as text files, or may be formatted rigidly. In general, a file is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user. The concept of a file is thus extremely general.

Because files are **the** method users and applications use to store and retrieve data, and because they are so general purpose, their use has stretched beyond its original confines. For example, UNIX, Linux, and some other operating systems provide a `proc` file system that uses file-system interfaces to provide access to system information (such as process details).

The information in a file is defined by its creator. Many different types of information may be stored in a file—source or executable programs, numeric or text data, photos, music, video, and so on. A file has a certain defined structure, which depends on its type. A **text file** is a sequence of characters organized into lines (and possibly pages). A **source file** is a sequence of functions, each of which is further organized as declarations followed by executable statements. An **executable file** is a series of code sections that the loader can bring into memory and execute.

### 13.1.1 File Attributes

A file is named, for the convenience of its human users, and is referred to by its name. A name is usually a string of characters, such as `example.c`. Some systems differentiate between uppercase and lowercase characters in names, whereas other systems do not. When a file is named, it becomes independent of the process, the user, and even the system that created it. For instance, one user might create the file `example.c`, and another user might edit that file by specifying its name. The file's owner might write the file to a USB drive, send it as an e-mail attachment, or copy it across a network, and it could still be called `example.c` on the destination system. Unless there is a sharing and synchonization method, that second copy is now independent of the first and can be changed separately.

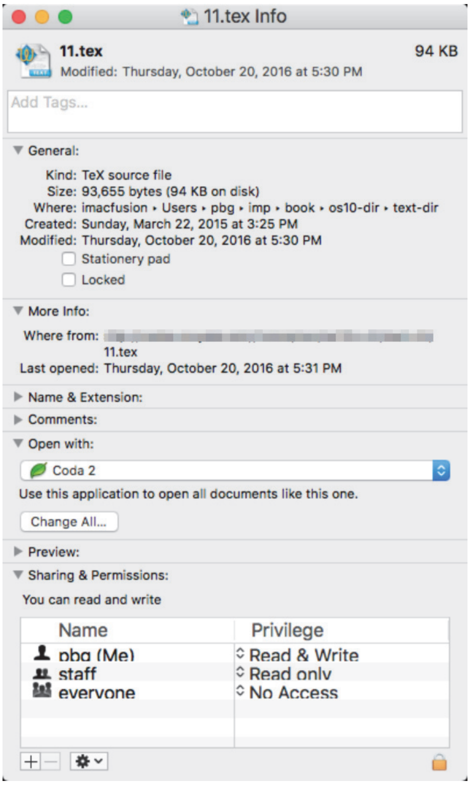
A file's attributes vary from one operating system to another but typically consist of these:

- **Name**. The symbolic file name is the only information kept in human-readable form.
- **Identifier**. This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for the file.
- **Type**. This information is needed for systems that support different types of files.
- **Location**. This information is a pointer to a device and to the location of the file on that device.

- **Size**. The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.
- **Protection**. Access-control information determines who can do reading, writing, executing, and so on.
- **Timestamps and user identification**. This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.

Some newer file systems also support **extended file attributes**, including character encoding of the file and security features such as a file checksum. Figure 13.1 illustrates a **file info window** on macOS that displays a file's attributes.

The information about all files is kept in the directory structure, which resides on the same device as the files themselves. Typically, a directory entry consists of the file's name and its unique identifier. The identifier in turn locates the other file attributes. It may take more than a kilobyte to record this information for each file. In a system with many files, the size of the directory itself may be megabytes or gigabytes. Because directories must match the volatility of the files, like files, they must be stored on the device and are usually brought into memory piecemeal, as needed.


11.tex Info

11.tex 94 KB
Modified: Thursday, October 20, 2016 at 5:30 PM

Add Tags...

General:

Kind: TeX source file
Size: 93,655 bytes (94 KB on disk)
Where: imacfusion ▸ Users ▸ pbg ▸ imp ▸ book ▸ os10-dir ▸ text-dir
Created: Sunday, March 22, 2015 at 3:25 PM
Modified: Thursday, October 20, 2016 at 5:30 PM
☐ Stationery pad
☐ Locked

More Info:

Where from: [redacted]
11.tex
Last opened: Thursday, October 20, 2016 at 5:31 PM

Name & Extension:

Comments:

Open with:

Coda 2

Use this application to open all documents like this one.

Change All...

Preview:

Sharing & Permissions:

You can read and write

| Name | Privilege |
|---|---|
| pbg (Me) | Read & Write |
| staff | Read only |
| everyone | No Access |

Figure 13.1 A file info window on macOS.

### 13.1.2 File Operations

A file is an abstract data type. To define a file properly, we need to consider the operations that can be performed on files. The operating system can provide system calls to create, write, read, reposition, delete, and truncate files. Let's examine what the operating system must do to perform each of these seven basic file operations. It should then be easy to see how other similar operations, such as renaming a file, can be implemented.

- **Creating a file**. Two steps are necessary to create a file. First, space in the file system must be found for the file. We discuss how to allocate space for the file in Chapter 14. Second, an entry for the new file must be made in a directory.
- **Opening a file**. Rather than have all file operations specify a file name, causing the operating system to evaluate the name, check access permissions, and so on, all operations except create and delete require a file `open()` first. If successful, the open call returns a file handle that is used as an argument in the other calls.
- **Writing a file**. To write a file, we make a system call specifying both the open file handle and the information to be written to the file. The system must keep a **write pointer** to the location in the file where the next write is to take place if it is sequential. The write pointer must be updated whenever a write occurs.
- **Reading a file**. To read from a file, we use a system call that specifies the file handle and where (in memory) the next block of the file should be put. Again, the system needs to keep a **read pointer** to the location in the file where the next read is to take place, if sequential. Once the read has taken place, the read pointer is updated. Because a process is usually either reading from or writing to a file, the current operation location can be kept as a per-process **current-file-position pointer**. Both the read and write operations use this same pointer, saving space and reducing system complexity.
- **Repositioning within a file**. The current-file-position pointer of the open file is repositioned to a given value. Repositioning within a file need not involve any actual I/O. This file operation is also known as a file **seek**.
- **Deleting a file**. To delete a file, we search the directory for the named file. Having found the associated directory entry, we release all file space, so that it can be reused by other files, and erase or mark as free the directory entry. Note that some systems allow **hard links**—multiple names (directory entries) for the same file. In this case the actual file contents is not deleted until the last link is deleted.
- **Truncating a file**. The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged—except for file length. The file can then be reset to length zero, and its file space can be released.

These seven basic operations comprise the minimal set of required file operations. Other common operations include appending new information to the end of an existing file and renaming an existing file. These primitive operations can then be combined to perform other file operations. For instance, we can create a copy of a file by creating a new file and then reading from the old and writing to the new. We also want to have operations that allow a user to get and set the various attributes of a file. For example, we may want to have operations that allow a user to determine the status of a file, such as the file's length, and to set file attributes, such as the file's owner.

As mentioned, most of the file operations mentioned involve searching the directory for the entry associated with the named file. To avoid this constant searching, many systems require that an `open()` system call be made before a file is first used. The operating system keeps a table, called the **open-file table**, containing information about all open files. When a file operation is requested, the file is specified via an index into this table, so no searching is required. When the file is no longer being actively used, it is closed by the process, and the operating system removes its entry from the open-file table, potentially releasing locks. `create()` and `delete()` are system calls that work with closed rather than open files.

Some systems implicitly open a file when the first reference to it is made. The file is automatically closed when the job or program that opened the file terminates. Most systems, however, require that the programmer open a file explicitly with the `open()` system call before that file can be used. The `open()` operation takes a file name and searches the directory, copying the directory entry into the open-file table. The `open()` call can also accept access-mode information—create, read-only, read−write, append-only, and so on. This mode is checked against the file's permissions. If the request mode is allowed, the file is opened for the process. The `open()` system call typically returns a pointer to the entry in the open-file table. This pointer, not the actual file name, is used in all I/O operations, avoiding any further searching and simplifying the system-call interface.

The implementation of the `open()` and `close()` operations is more complicated in an environment where several processes may open the file simultaneously. This may occur in a system where several different applications open the same file at the same time. Typically, the operating system uses two levels of internal tables: a per-process table and a system-wide table. The per-process table tracks all files that a process has open. Stored in this table is information regarding the process's use of the file. For instance, the current file pointer for each file is found here. Access rights to the file and accounting information can also be included.

Each entry in the per-process table in turn points to a system-wide open-file table. The system-wide table contains process-independent information, such as the location of the file on disk, access dates, and file size. Once a file has been opened by one process, the system-wide table includes an entry for the file. When another process executes an `open()` call, a new entry is simply added to the process's open-file table pointing to the appropriate entry in the system-wide table. Typically, the open-file table also has an **open count** associated with each file to indicate how many processes have the file open. Each `close()` decreases this open count, and when the open count reaches zero, the file is no longer in use, and the file's entry is removed from the open-file table.

### *FILE LOCKING IN JAVA*

In the Java API, acquiring a lock requires first obtaining the `FileChannel` for the file to be locked. The `lock()` method of the `FileChannel` is used to acquire the lock. The API of the `lock()` method is

```
FileLock lock(long begin, long end, boolean shared)
```

where `begin` and `end` are the beginning and ending positions of the region being locked. Setting `shared` to `true` is for shared locks; setting `shared` to `false` acquires the lock exclusively. The lock is released by invoking the `release()` of the `FileLock` returned by the `lock()` operation.

The program in Figure 13.2 illustrates file locking in Java. This program acquires two locks on the file `file.txt`. The lock for the first half of the file is an exclusive lock; the lock for the second half is a shared lock.

In summary, several pieces of information are associated with an open file.

- **File pointer**. On systems that do not include a file offset as part of the `read()` and `write()` system calls, the system must track the last read–write location as a current-file-position pointer. This pointer is unique to each process operating on the file and therefore must be kept separate from the on-disk file attributes.
- **File-open count**. As files are closed, the operating system must reuse its open-file table entries, or it could run out of space in the table. Multiple processes may have opened a file, and the system must wait for the last file to close before removing the open-file table entry. The file-open count tracks the number of opens and closes and reaches zero on the last close. The system can then remove the entry.
- **Location of the file**. Most file operations require the system to read or write data within the file. The information needed to locate the file (wherever it is located, be it on mass storage, on a file server across the network, or on a RAM drive) is kept in memory so that the system does not have to read it from the directory structure for each operation.
- **Access rights**. Each process opens a file in an access mode. This information is stored on the per-process table so the operating system can allow or deny subsequent I/O requests.

Some operating systems provide facilities for locking an open file (or sections of a file). File locks allow one process to lock a file and prevent other processes from gaining access to it. File locks are useful for files that are shared by several processes—for example, a system log file that can be accessed and modified by a number of processes in the system.

File locks provide functionality similar to reader–writer locks, covered in Section 7.1.2. A **shared lock** is akin to a reader lock in that several processes can acquire the lock concurrently. An **exclusive lock** behaves like a writer lock; only one process at a time can acquire such a lock. It is important to note that not

```
import java.io.*;
import java.nio.channels.*;

public class LockingExample {
 public static final boolean EXCLUSIVE = false;
 public static final boolean SHARED = true;

 public static void main(String args[]) throws IOException {
  FileLock sharedLock = null;
  FileLock exclusiveLock = null;

  try {
   RandomAccessFile raf = new RandomAccessFile("file.txt","rw");

   // get the channel for the file
   FileChannel ch = raf.getChannel();

   // this locks the first half of the file - exclusive
   exclusiveLock = ch.lock(0, raf.length()/2, EXCLUSIVE);

   /** Now modify the data . . . */

   // release the lock
   exclusiveLock.release();

   // this locks the second half of the file - shared
   sharedLock = ch.lock(raf.length()/2+1,raf.length(),SHARED);

   /** Now read the data . . . */

   // release the lock
   sharedLock.release();
  } catch (java.io.IOException ioe) {
   System.err.println(ioe);
  }
  finally {
   if (exclusiveLock != null)
        exclusiveLock.release();
   if (sharedLock != null)
        sharedLock.release();
  }
 }
}
```

**Figure 13.2** File-locking example in Java.

all operating systems provide both types of locks: some systems provide only exclusive file locking.

Furthermore, operating systems may provide either **mandatory** or **advisory** file-locking mechanisms. With mandatory locking, once a process acquires an exclusive lock, the operating system will prevent any other process from accessing the locked file. For example, assume a process acquires an exclusive lock on the file `system.log`. If we attempt to open `system.log` from another process—for example, a text editor—the operating system will prevent access until the exclusive lock is released. Alternatively, if the lock is advisory, then the operating system will not prevent the text editor from acquiring access to `system.log`. Rather, the text editor must be written so that it manually acquires the lock before accessing the file. In other words, if the locking scheme is mandatory, the operating system ensures locking integrity. For advisory locking, it is up to software developers to ensure that locks are appropriately acquired and released. As a general rule, Windows operating systems adopt mandatory locking, and UNIX systems employ advisory locks.

The use of file locks requires the same precautions as ordinary process synchronization. For example, programmers developing on systems with mandatory locking must be careful to hold exclusive file locks only while they are accessing the file. Otherwise, they will prevent other processes from accessing the file as well. Furthermore, some measures must be taken to ensure that two or more processes do not become involved in a deadlock while trying to acquire file locks.

### 13.1.3 File Types

When we design a file system—indeed, an entire operating system—we always consider whether the operating system should recognize and support file types. If an operating system recognizes the type of a file, it can then operate on the file in reasonable ways. For example, a common mistake occurs when a user tries to output the binary-object form of a program. This attempt normally produces garbage; however, the attempt can succeed if the operating system has been told that the file is a binary-object program.

A common technique for implementing file types is to include the type as part of the file name. The name is split into two parts—a name and an extension, usually separated by a period (Figure 13.3). In this way, the user and the operating system can tell from the name alone what the type of a file is. Most operating systems allow users to specify a file name as a sequence of characters followed by a period and terminated by an extension made up of additional characters. Examples include `resume.docx`, `server.c`, and `ReaderThread.cpp`.

The system uses the extension to indicate the type of the file and the type of operations that can be done on that file. Only a file with a `.com`, `.exe`, or `.sh` extension can be executed, for instance. The `.com` and `.exe` files are two forms of binary executable files, whereas the `.sh` file is a **shell script** containing, in ASCII format, commands to the operating system. Application programs also use extensions to indicate file types in which they are interested. For example, Java compilers expect source files to have a `.java` extension, and the Microsoft Word word processor expects its files to end with a `.doc` or `.docx` extension. These extensions are not always required, so a user may specify a file without the extension (to save typing), and the application will look for a file with the given name and the extension it expects. Because these extensions are not supported by the operating system, they can be considered “hints” to the applications that operate on them.

Consider, too, the macOS operating system. In this system, each file has a type, such as `.app` (for application). Each file also has a creator attribute

| file type | usual extension | function |
|---|---|---|
| executable | exe, com, bin or none | ready-to-run machine-language program |
| object | obj, o | compiled, machine language, not linked |
| source code | c, cc, java, perl, asm | source code in various languages |
| batch | bat, sh | commands to the command interpreter |
| markup | xml, html, tex | textual data, documents |
| word processor | xml, rtf, docx | various word-processor formats |
| library | lib, a, so, dll | libraries of routines for programmers |
| print or view | gif, pdf, jpg | ASCII or binary file in a format for printing or viewing |
| archive | rar, zip, tar | related files grouped into one file, sometimes compressed, for archiving or storage |
| multimedia | mpeg, mov, mp3, mp4, avi | binary file containing audio or A/V information |

**Figure 13.3** Common file types.

containing the name of the program that created it. This attribute is set by the operating system during the `create()` call, so its use is enforced and supported by the system. For instance, a file produced by a word processor has the word processor's name as its creator. When the user opens that file, by double-clicking the mouse on the icon representing the file, the word processor is invoked automatically, and the file is loaded, ready to be edited.

The UNIX system uses a **magic number** stored at the beginning of some binary files to indicate the type of data in the file (for example, the format of an image file). Likewise, it uses a text magic number at the start of text files to indicate the type of file (which shell language a script is written in) and so on. (For more details on magic numbers and other computer jargon, see http://www.catb.org/esr/jargon/.) Not all files have magic numbers, so system features cannot be based solely on this information. UNIX does not record the name of the creating program, either. UNIX does allow file-name-extension hints, but these extensions are neither enforced nor depended on by the operating system; they are meant mostly to aid users in determining what type of contents the file contains. Extensions can be used or ignored by a given application, but that is up to the application's programmer.

### 13.1.4 File Structure

File types also can be used to indicate the internal structure of the file. Source and object files have structures that match the expectations of the programs that read them. Further, certain files must conform to a required structure that is understood by the operating system. For example, the operating system requires that an executable file have a specific structure so that it can determine where in memory to load the file and what the location of the first instruction is. Some operating systems extend this idea into a set of system-supported file structures, with sets of special operations for manipulating files with those structures.

This point brings us to one of the disadvantages of having the operating system support multiple file structures: it makes the operating system large and cumbersome. If the operating system defines five different file structures, it needs to contain the code to support these file structures. In addition, it may be necessary to define every file as one of the file types supported by the operating system. When new applications require information structured in ways not supported by the operating system, severe problems may result.

For example, assume that a system supports two types of files: text files (composed of ASCII characters separated by a carriage return and line feed) and executable binary files. Now, if we (as users) want to define an encrypted file to protect the contents from being read by unauthorized people, we may find neither file type to be appropriate. The encrypted file is not ASCII text lines but rather is (apparently) random bits. Although it may appear to be a binary file, it is not executable. As a result, we may have to circumvent or misuse the operating system's file-type mechanism or abandon our encryption scheme.

Some operating systems impose (and support) a minimal number of file structures. This approach has been adopted in UNIX, Windows, and others. UNIX considers each file to be a sequence of 8-bit bytes; no interpretation of these bits is made by the operating system. This scheme provides maximum flexibility but little support. Each application program must include its own code to interpret an input file as to the appropriate structure. However, all operating systems must support at least one structure—that of an executable file—so that the system is able to load and run programs.

### 13.1.5 Internal File Structure

Internally, locating an offset within a file can be complicated for the operating system. Disk systems typically have a well-defined block size determined by the size of a sector. All disk I/O is performed in units of one block (physical record), and all blocks are the same size. It is unlikely that the physical record size will exactly match the length of the desired logical record. Logical records may even vary in length. Packing a number of logical records into physical blocks is a common solution to this problem.

For example, the UNIX operating system defines all files to be simply streams of bytes. Each byte is individually addressable by its offset from the beginning (or end) of the file. In this case, the logical record size is 1 byte. The file system automatically packs and unpacks bytes into physical disk blocks—say, 512 bytes per block—as necessary.

The logical record size, physical block size, and packing technique determine how many logical records are in each physical block. The packing can be done either by the user's application program or by the operating system. In either case, the file may be considered a sequence of blocks. All the basic I/O functions operate in terms of blocks. The conversion from logical records to physical blocks is a relatively simple software problem.

Because disk space is always allocated in blocks, some portion of the last block of each file is generally wasted. If each block were 512 bytes, for example, then a file of 1,949 bytes would be allocated four blocks (2,048 bytes); the last 99 bytes would be wasted. The waste incurred to keep everything in units of blocks (instead of bytes) is internal fragmentation. All file systems suffer from internal fragmentation; the larger the block size, the greater the internal fragmentation.

## 13.2 Access Methods

Files store information. When it is used, this information must be accessed and read into computer memory. The information in the file can be accessed in several ways. Some systems provide only one access method for files. Others (such as mainframe operating systems) support many access methods, and choosing the right one for a particular application is a major design problem.

### 13.2.1 Sequential Access

The simplest access method is **sequential access**. Information in the file is processed in order, one record after the other. This mode of access is by far the most common; for example, editors and compilers usually access files in this fashion.

Reads and writes make up the bulk of the operations on a file. A read operation—`read_next()`—reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location. Similarly, the write operation—`write_next()`—appends to the end of the file and advances to the end of the newly written material (the new end of file). Such a file can be reset to the beginning, and on some systems, a program may be able to skip forward or backward $n$ records for some integer $n$—perhaps only for $n = 1$. Sequential access, which is depicted in Figure 13.4, is based on a tape model of a file and works as well on sequential-access devices as it does on random-access ones.

### 13.2.2 Direct Access

Another method is **direct access** (or **relative access**). Here, a file is made up of fixed-length **logical records** that allow programs to read and write records rapidly in no particular order. The direct-access method is based on a disk model of a file, since disks allow random access to any file block. For direct access, the file is viewed as a numbered sequence of blocks or records. Thus,

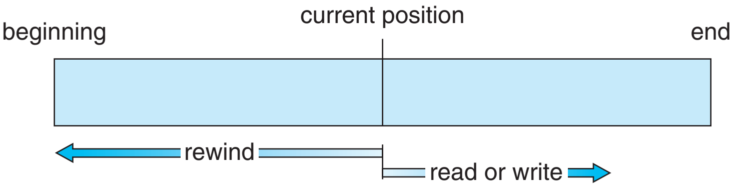


**Figure 13.4** Sequential-access file.

we may read block 14, then read block 53, and then write block 7. There are no restrictions on the order of reading or writing for a direct-access file.

Direct-access files are of great use for immediate access to large amounts of information. Databases are often of this type. When a query concerning a particular subject arrives, we compute which block contains the answer and then read that block directly to provide the desired information.

As a simple example, on an airline-reservation system, we might store all the information about a particular flight (for example, flight 713) in the block identified by the flight number. Thus, the number of available seats for flight 713 is stored in block 713 of the reservation file. To store information about a larger set, such as people, we might compute a hash function on the people's names or search a small in-memory index to determine a block to read and search.

For the direct-access method, the file operations must be modified to include the block number as a parameter. Thus, we have `read(n)`, where *n* is the block number, rather than `read_next()`, and `write(n)` rather than `write_next()`. An alternative approach is to retain `read_next()` and `write_next()` and to add an operation `position_file(n)` where *n* is the block number. Then, to effect a `read(n)`, we would `position_file(n)` and then `read_next()`.

The block number provided by the user to the operating system is normally a **relative block number**. A relative block number is an index relative to the beginning of the file. Thus, the first relative block of the file is 0, the next is 1, and so on, even though the absolute disk address may be 14703 for the first block and 3192 for the second. The use of relative block numbers allows the operating system to decide where the file should be placed (called the **allocation problem**, as we discuss in Chapter 14) and helps to prevent the user from accessing portions of the file system that may not be part of her file. Some systems start their relative block numbers at 0; others start at 1.

How, then, does the system satisfy a request for record $N$ in a file? Assuming we have a logical record length $L$, the request for record $N$ is turned into an I/O request for $L$ bytes starting at location $L * (N)$ within the file (assuming the first record is $N = 0$). Since logical records are of a fixed size, it is also easy to read, write, or delete a record.

Not all operating systems support both sequential and direct access for files. Some systems allow only sequential file access; others allow only direct access. Some systems require that a file be defined as sequential or direct when it is created. Such a file can be accessed only in a manner consistent with its declaration. We can easily simulate sequential access on a direct-access file by simply keeping a variable *cp* that defines our current position, as shown in Figure 13.5. Simulating a direct-access file on a sequential-access file, however, is extremely inefficient and clumsy.

### 13.2.3 Other Access Methods

Other access methods can be built on top of a direct-access method. These methods generally involve the construction of an index for the file. The **index**, like an index in the back of a book, contains pointers to the various blocks. To find a record in the file, we first search the index and then use the pointer to access the file directly and to find the desired record.

| sequential access | implementation for direct access |
|---|---|
| reset | cp = 0; |
| read_next | read cp;<br>cp = cp + 1; |
| write_next | write cp;<br>cp = cp + 1; |

**Figure 13.5** Simulation of sequential access on a direct-access file.

For example, a retail-price file might list the universal product codes (UPCs) for items, with the associated prices. Each record consists of a 10-digit UPC and a 6-digit price, for a 16-byte record. If our disk has 1,024 bytes per block, we can store 64 records per block. A file of 120,000 records would occupy about 2,000 blocks (2 million bytes). By keeping the file sorted by UPC, we can define an index consisting of the first UPC in each block. This index would have 2,000 entries of 10 digits each, or 20,000 bytes, and thus could be kept in memory. To find the price of a particular item, we can make a binary search of the index. From this search, we learn exactly which block contains the desired record and access that block. This structure allows us to search a large file doing little I/O.

With large files, the index file itself may become too large to be kept in memory. One solution is to create an index for the index file. The primary index file contains pointers to secondary index files, which point to the actual data items.

For example, IBM's indexed sequential-access method (ISAM) uses a small master index that points to disk blocks of a secondary index. The secondary index blocks point to the actual file blocks. The file is kept sorted on a defined key. To find a particular item, we first make a binary search of the master index, which provides the block number of the secondary index. This block is read in, and again a binary search is used to find the block containing the desired record. Finally, this block is searched sequentially. In this way, any record can be located from its key by at most two direct-access reads. Figure 13.6 shows a similar situation as implemented by OpenVMS index and relative files.

## 13.3 Directory Structure

The directory can be viewed as a symbol table that translates file names into their file control blocks. If we take such a view, we see that the directory itself can be organized in many ways. The organization must allow us to insert entries, to delete entries, to search for a named entry, and to list all the entries in the directory. In this section, we examine several schemes for defining the logical structure of the directory system.

When considering a particular directory structure, we need to keep in mind the operations that are to be performed on a directory:

- **Search for a file**. We need to be able to search a directory structure to find the entry for a particular file. Since files have symbolic names, and similar

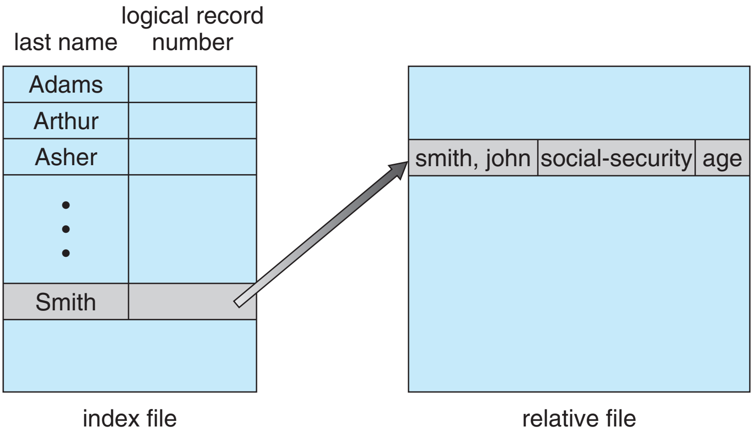


**Figure 13.6** Example of index and relative files.

names may indicate a relationship among files, we may want to be able to find all files whose names match a particular pattern.

- **Create a file**. New files need to be created and added to the directory.
- **Delete a file**. When a file is no longer needed, we want to be able to remove it from the directory. Note a delete leaves a hole in the directory structure and the file system may have a method to defragement the directory structure.
- **List a directory**. We need to be able to list the files in a directory and the contents of the directory entry for each file in the list.
- **Rename a file**. Because the name of a file represents its contents to its users, we must be able to change the name when the contents or use of the file changes. Renaming a file may also allow its position within the directory structure to be changed.
- **Traverse the file system**. We may wish to access every directory and every file within a directory structure. For reliability, it is a good idea to save the contents and structure of the entire file system at regular intervals. Often, we do this by copying all files to magnetic tape, other secondary storage, or across a network to another system or the cloud. This technique provides a backup copy in case of system failure. In addition, if a file is no longer in use, the file can be copied the backup target and the disk space of that file released for reuse by another file.

In the following sections, we describe the most common schemes for defining the logical structure of a directory.

### 13.3.1 Single-Level Directory

The simplest directory structure is the single-level directory. All files are contained in the same directory, which is easy to support and understand (Figure 13.7).

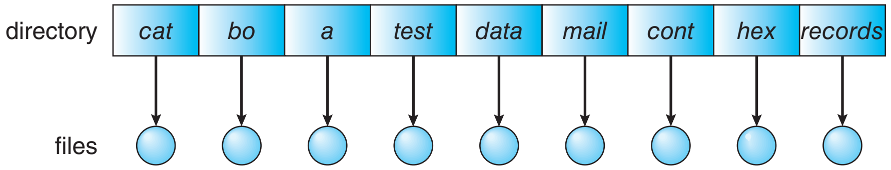


**Figure 13.7** Single-level directory.

A single-level directory has significant limitations, however, when the number of files increases or when the system has more than one user. Since all files are in the same directory, they must have unique names. If two users call their data file `test.txt`, then the unique-name rule is violated. For example, in one programming class, 23 students called the program for their second assignment `prog2.c`; another 11 called it `assign2.c`. Fortunately, most file systems support file names of up to 255 characters, so it is relatively easy to select unique file names.

Even a single user on a single-level directory may find it difficult to remember the names of all the files as the number of files increases. It is not uncommon for a user to have hundreds of files on one computer system and an equal number of additional files on another system. Keeping track of so many files is a daunting task.

### 13.3.2 Two-Level Directory

As we have seen, a single-level directory often leads to confusion of file names among different users. The standard solution is to create a separate directory for each user.

In the two-level directory structure, each user has his own **user file directory** (**UFD**). The UFDs have similar structures, but each lists only the files of a single user. When a user job starts or a user logs in, the system's **master file directory** (**MFD**) is searched. The MFD is indexed by user name or account number, and each entry points to the UFD for that user (Figure 13.8).

When a user refers to a particular file, only his own UFD is searched. Thus, different users may have files with the same name, as long as all the file names within each UFD are unique. To create a file for a user, the operating system searches only that user's UFD to ascertain whether another file of that name

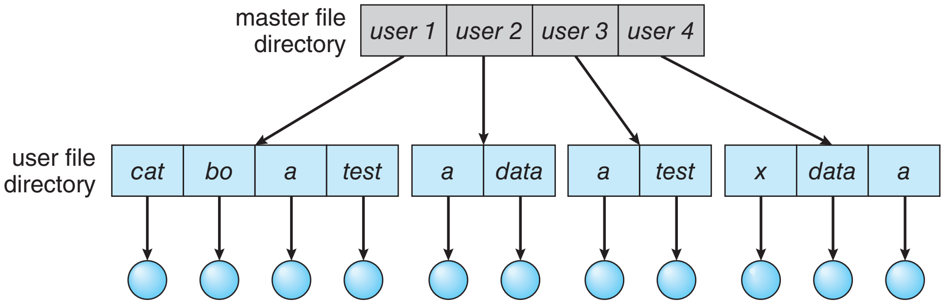


**Figure 13.8** Two-level directory structure.

exists. To delete a file, the operating system confines its search to the local UFD; thus, it cannot accidentally delete another user's file that has the same name.

The user directories themselves must be created and deleted as necessary. A special system program is run with the appropriate user name and account information. The program creates a new UFD and adds an entry for it to the MFD. The execution of this program might be restricted to system administrators. The allocation of disk space for user directories can be handled with the techniques discussed in Chapter 14 for files themselves.

Although the two-level directory structure solves the name-collision problem, it still has disadvantages. This structure effectively isolates one user from another. Isolation is an advantage when the users are completely independent but is a disadvantage when the users want to cooperate on some task and to access one another's files. Some systems simply do not allow local user files to be accessed by other users.

If access is to be permitted, one user must have the ability to name a file in another user's directory. To name a particular file uniquely in a two-level directory, we must give both the user name and the file name. A two-level directory can be thought of as a tree, or an inverted tree, of height 2. The root of the tree is the MFD. Its direct descendants are the UFDs. The descendants of the UFDs are the files themselves. The files are the leaves of the tree. Specifying a user name and a file name defines a path in the tree from the root (the MFD) to a leaf (the specified file). Thus, a user name and a file name define a **path name**. Every file in the system has a path name. To name a file uniquely, a user must know the path name of the file desired.

For example, if user A wishes to access her own test file named `test.txt`, she can simply refer to `test.txt`. To access the file named `test.txt` of user B (with directory-entry name `userb`), however, she might have to refer to `/userb/test.txt`. Every system has its own syntax for naming files in directories other than the user's own.

Additional syntax is needed to specify the volume of a file. For instance, in Windows a volume is specified by a letter followed by a colon. Thus, a file specification might be `C:\userb\test`. Some systems go even further and separate the volume, directory name, and file name parts of the specification. In OpenVMS, for instance, the file `login.com` might be specified as: `u:[sst.crissmeyer]login.com;1`, where `u` is the name of the volume, `sst` is the name of the directory, `crissmeyer` is the name of the subdirectory, and `1` is the version number. Other systems—such as UNIX and Linux—simply treat the volume name as part of the directory name. The first name given is that of the volume, and the rest is the directory and file. For instance, `/u/pgalvin/test` might specify volume `u`, directory `pgalvin`, and file `test`.

A special instance of this situation occurs with the system files. Programs provided as part of the system—loaders, assemblers, compilers, utility routines, libraries, and so on—are generally defined as files. When the appropriate commands are given to the operating system, these files are read by the loader and executed. Many command interpreters simply treat such a command as the name of a file to load and execute. In the directory system as we defined it above, this file name would be searched for in the current UFD. One solution would be to copy the system files into each UFD. However, copying all the system files would waste an enormous amount of space. (If the system files require 5 MB, then supporting 12 users would require 5 × 12 = 60 MB just for copies of the system files.)

The standard solution is to complicate the search procedure slightly. A special user directory is defined to contain the system files (for example, user 0). Whenever a file name is given to be loaded, the operating system first searches the local UFD. If the file is found, it is used. If it is not found, the system automatically searches the special user directory that contains the system files. The sequence of directories searched when a file is named is called the **search path**. The search path can be extended to contain an unlimited list of directories to search when a command name is given. This method is the one most used in UNIX and Windows. Systems can also be designed so that each user has his own search path.

### 13.3.3 Tree-Structured Directories

Once we have seen how to view a two-level directory as a two-level tree, the natural generalization is to extend the directory structure to a tree of arbitrary height (Figure 13.9). This generalization allows users to create their own subdirectories and to organize their files accordingly. A tree is the most common directory structure. The tree has a root directory, and every file in the system has a unique path name.

A directory (or subdirectory) contains a set of files or subdirectories. In many implementations, a directory is simply another file, but it is treated in a special way. All directories have the same internal format. One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1). Special

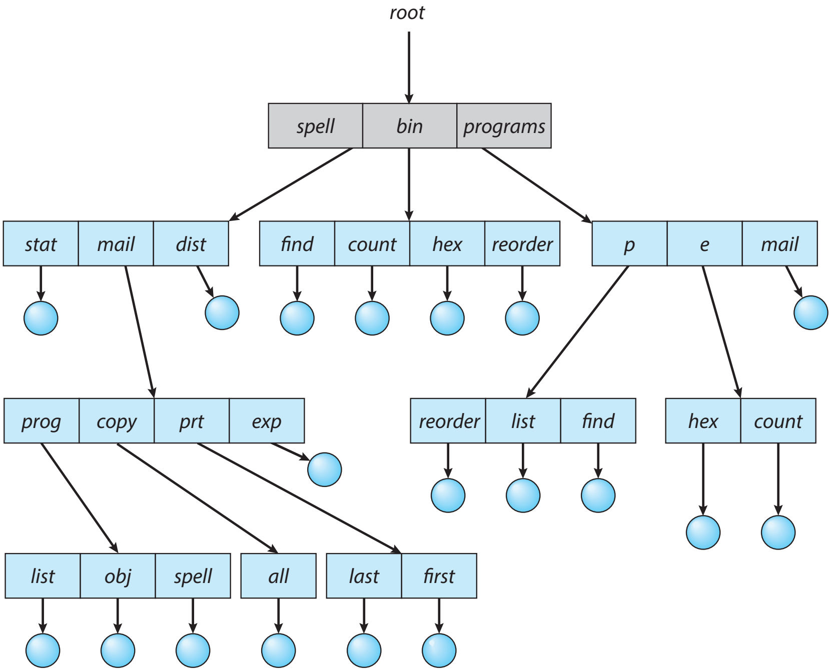


**Figure 13.9** Tree-structured directory structure.

system calls are used to create and delete directories. In this case the operating system (or the file system code) implements another file format, that of a directory.

In normal use, each process has a current directory. The **current directory** should contain most of the files that are of current interest to the process. When reference is made to a file, the current directory is searched. If a file is needed that is not in the current directory, then the user usually must either specify a path name or change the current directory to be the directory holding that file. To change directories, a system call could be provided that takes a directory name as a parameter and uses it to redefine the current directory. Thus, the user can change her current directory whenever she wants. Other systems leave it to the application (say, a shell) to track and operate on a current directory, as each process could have different current directories.

The initial current directory of a user's login shell is designated when the user job starts or the user logs in. The operating system searches the accounting file (or some other predefined location) to find an entry for this user (for accounting purposes). In the accounting file is a pointer to (or the name of) the user's initial directory. This pointer is copied to a local variable for this user that specifies the user's initial current directory. From that shell, other processes can be spawned. The current directory of any subprocess is usually the current directory of the parent when it was spawned.

Path names can be of two types: absolute and relative. In UNIX and Linux, an **absolute path name** begins at the root (which is designated by an initial "/") and follows a path down to the specified file, giving the directory names on the path. A **relative path name** defines a path from the current directory. For example, in the tree-structured file system of Figure 13.9, if the current directory is `/spell/mail`, then the relative path name `prt/first` refers to the same file as does the absolute path name `/spell/mail/prt/first`.

Allowing a user to define her own subdirectories permits her to impose a structure on her files. This structure might result in separate directories for files associated with different topics (for example, a subdirectory was created to hold the text of this book) or different forms of information. For example, the directory `programs` may contain source programs; the directory `bin` may store all the binaries. (As a side note, executable files were known in many systems as "binaries" which led to them being stored in the `bin` directory.)

An interesting policy decision in a tree-structured directory concerns how to handle the deletion of a directory. If a directory is empty, its entry in the directory that contains it can simply be deleted. However, suppose the directory to be deleted is not empty but contains several files or subdirectories. One of two approaches can be taken. Some systems will not delete a directory unless it is empty. Thus, to delete a directory, the user must first delete all the files in that directory. If any subdirectories exist, this procedure must be applied recursively to them, so that they can be deleted also. This approach can result in a substantial amount of work. An alternative approach, such as that taken by the UNIX `rm` command, is to provide an option: when a request is made to delete a directory, all that directory's files and subdirectories are also to be deleted. Either approach is fairly easy to implement; the choice is one of policy. The latter policy is more convenient, but it is also more dangerous, because an entire directory structure can be removed with one command. If that command is issued in error, a large number of files and directories will need to be restored (assuming a backup exists).

With a tree-structured directory system, users can be allowed to access, in addition to their files, the files of other users. For example, user B can access a file of user A by specifying its path name. User B can specify either an absolute or a relative path name. Alternatively, user B can change her current directory to be user A's directory and access the file by its file name.

### 13.3.4 Acyclic-Graph Directories

Consider two programmers who are working on a joint project. The files associated with that project can be stored in a subdirectory, separating them from other projects and files of the two programmers. But since both programmers are equally responsible for the project, both want the subdirectory to be in their own directories. In this situation, the common subdirectory should be ***shared.*** A shared directory or file exists in the file system in two (or more) places at once.

A tree structure prohibits the sharing of files or directories. An **acyclic graph**—that is, a graph with no cycles—allows directories to share subdirectories and files (Figure 13.10). The same file or subdirectory may be in two different directories. The acyclic graph is a natural generalization of the tree-structured directory scheme.

It is important to note that a shared file (or directory) is not the same as two copies of the file. With two copies, each programmer can view the copy rather than the original, but if one programmer changes the file, the changes will not appear in the other's copy. With a shared file, only one actual file exists, so any changes made by one person are immediately visible to the other. Sharing is

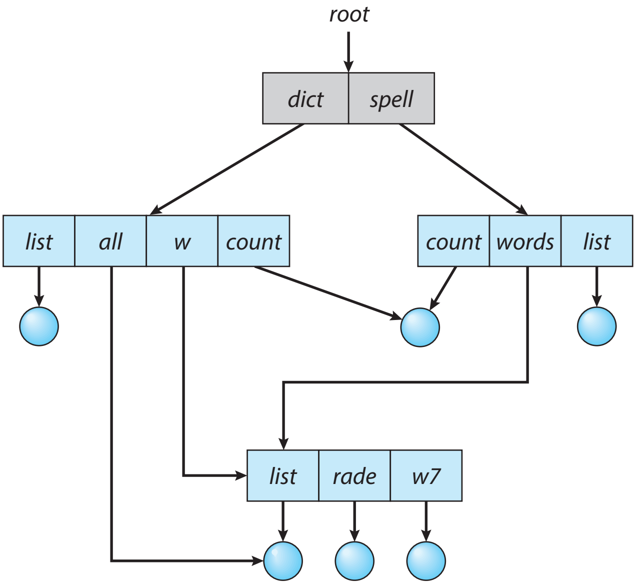


**Figure 13.10** Acyclic-graph directory structure.

particularly important for subdirectories; a new file created by one person will automatically appear in all the shared subdirectories.

When people are working as a team, all the files they want to share can be put into one directory. The home directory of each team member could contain this directory of shared files as a subdirectory. Even in the case of a single user, the user's file organization may require that some file be placed in different subdirectories. For example, a program written for a particular project should be both in the directory of all programs and in the directory for that project.

Shared files and subdirectories can be implemented in several ways. A common way, exemplified by UNIX systems, is to create a new directory entry called a link. A **link** is effectively a pointer to another file or subdirectory. For example, a link may be implemented as an absolute or a relative path name. When a reference to a file is made, we search the directory. If the directory entry is marked as a link, then the name of the real file is included in the link information. We **resolve** the link by using that path name to locate the real file. Links are easily identified by their format in the directory entry (or by having a special type on systems that support types) and are effectively indirect pointers. The operating system ignores these links when traversing directory trees to preserve the acyclic structure of the system.

Another common approach to implementing shared files is simply to duplicate all information about them in both sharing directories. Thus, both entries are identical and equal. Consider the difference between this approach and the creation of a link. The link is clearly different from the original directory entry; thus, the two are not equal. Duplicate directory entries, however, make the original and the copy indistinguishable. A major problem with duplicate directory entries is maintaining consistency when a file is modified.

An acyclic-graph directory structure is more flexible than a simple tree structure, but it is also more complex. Several problems must be considered carefully. A file may now have multiple absolute path names. Consequently, distinct file names may refer to the same file. This situation is similar to the aliasing problem for programming languages. If we are trying to traverse the entire file system—to find a file, to accumulate statistics on all files, or to copy all files to backup storage—this problem becomes significant, since we do not want to traverse shared structures more than once.

Another problem involves deletion. When can the space allocated to a shared file be deallocated and reused? One possibility is to remove the file whenever anyone deletes it, but this action may leave dangling pointers to the now-nonexistent file. Worse, if the remaining file pointers contain actual disk addresses, and the space is subsequently reused for other files, these dangling pointers may point into the middle of other files.

In a system where sharing is implemented by symbolic links, this situation is somewhat easier to handle. The deletion of a link need not affect the original file; only the link is removed. If the file entry itself is deleted, the space for the file is deallocated, leaving the links dangling. We can search for these links and remove them as well, but unless a list of the associated links is kept with each file, this search can be expensive. Alternatively, we can leave the links until an attempt is made to use them. At that time, we can determine that the file of the name given by the link does not exist and can fail to resolve the link name; the access is treated just as with any other illegal file name. (In this case, the system designer should consider carefully what to do when a file is deleted and another file of the same name is created, before a symbolic link to the original file is used.) In the case of UNIX, symbolic links are left when a file is deleted, and it is up to the user to realize that the original file is gone or has been replaced. Microsoft Windows uses the same approach.

Another approach to deletion is to preserve the file until all references to it are deleted. To implement this approach, we must have some mechanism for determining that the last reference to the file has been deleted. We could keep a list of all references to a file (directory entries or symbolic links). When a link or a copy of the directory entry is established, a new entry is added to the file-reference list. When a link or directory entry is deleted, we remove its entry on the list. The file is deleted when its file-reference list is empty.

The trouble with this approach is the variable and potentially large size of the file-reference list. However, we really do not need to keep the entire list—we need to keep only a count of the number of references. Adding a new link or directory entry increments the reference count. Deleting a link or entry decrements the count. When the count is 0, the file can be deleted; there are no remaining references to it. The UNIX operating system uses this approach for nonsymbolic links (or **hard links**), keeping a reference count in the file information block (or inode; see Section C.7.2). By effectively prohibiting multiple references to directories, we maintain an acyclic-graph structure.

To avoid problems such as the ones just discussed, some systems simply do not allow shared directories or links.

### 13.3.5 General Graph Directory

A serious problem with using an acyclic-graph structure is ensuring that there are no cycles. If we start with a two-level directory and allow users to create subdirectories, a tree-structured directory results. It should be fairly easy to see that simply adding new files and subdirectories to an existing tree-structured directory preserves the tree-structured nature. However, when we add links, the tree structure is destroyed, resulting in a simple graph structure (Figure 13.11).

The primary advantage of an acyclic graph is the relative simplicity of the algorithms to traverse the graph and to determine when there are no more references to a file. We want to avoid traversing shared sections of an acyclic graph twice, mainly for performance reasons. If we have just searched a major shared subdirectory for a particular file without finding it, we want to avoid searching that subdirectory again; the second search would be a waste of time.

If cycles are allowed to exist in the directory, we likewise want to avoid searching any component twice, for reasons of correctness as well as performance. A poorly designed algorithm might result in an infinite loop continually searching through the cycle and never terminating. One solution is to limit arbitrarily the number of directories that will be accessed during a search.

A similar problem exists when we are trying to determine when a file can be deleted. With acyclic-graph directory structures, a value of 0 in the reference count means that there are no more references to the file or directory, and the file can be deleted. However, when cycles exist, the reference count may not be 0 even when it is no longer possible to refer to a directory or file. This anomaly results from the possibility of self-referencing (or a cycle) in the directory structure. In this case, we generally need to use a **garbage collection**

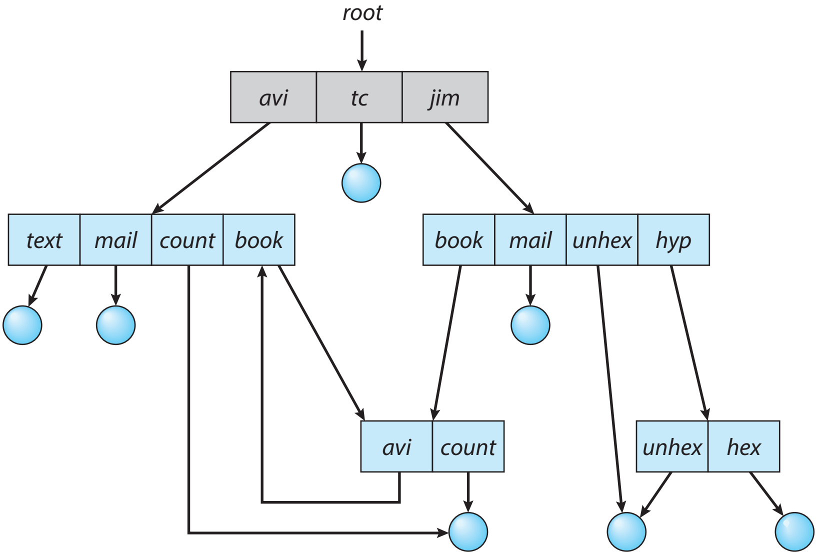


**Figure 13.11** General graph directory.

scheme to determine when the last reference has been deleted and the disk space can be reallocated. Garbage collection involves traversing the entire file system, marking everything that can be accessed. Then, a second pass collects everything that is not marked onto a list of free space. (A similar marking procedure can be used to ensure that a traversal or search will cover everything in the file system once and only once.) Garbage collection for a disk-based file system, however, is extremely time consuming and is thus seldom attempted.

Garbage collection is necessary only because of possible cycles in the graph. Thus, an acyclic-graph structure is much easier to work with. The difficulty is to avoid cycles as new links are added to the structure. How do we know when a new link will complete a cycle? There are algorithms to detect cycles in graphs; however, they are computationally expensive, especially when the graph is on disk storage. A simpler algorithm in the special case of directories and links is to bypass links during directory traversal. Cycles are avoided, and no extra overhead is incurred.

## 13.4 Protection

When information is stored in a computer system, we want to keep it safe from physical damage (the issue of reliability) and improper access (the issue of protection).

Reliability is generally provided by duplicate copies of files. Many computers have systems programs that automatically (or through computer-operator intervention) copy disk files to tape at regular intervals (once per day or week or month) to maintain a copy should a file system be accidentally destroyed. File systems can be damaged by hardware problems (such as errors in reading or writing), power surges or failures, head crashes, dirt, temperature extremes, and vandalism. Files may be deleted accidentally. Bugs in the file-system software can also cause file contents to be lost. Reliability was covered in more detail in Chapter 11.

Protection can be provided in many ways. For a laptop system running a modern operating system, we might provide protection by requiring a user name and password authentication to access it, encrypting the secondary storage so even someone opening the laptop and removing the drive would have a difficult time accessing its data, and firewalling network access so that when it is in use it is difficult to break in via its network connection. In multiuser system, even valid access of the system needs more advanced mechanisms to allow only valid access of the data.

### 13.4.1 Types of Access

The need to protect files is a direct result of the ability to access files. Systems that do not permit access to the files of other users do not need protection. Thus, we could provide complete protection by prohibiting access. Alternatively, we could provide free access with no protection. Both approaches are too extreme for general use. What is needed is controlled access.

Protection mechanisms provide controlled access by limiting the types of file access that can be made. Access is permitted or denied depending on several factors, one of which is the type of access requested. Several different types of operations may be controlled:

- **Read**. Read from the file.
- **Write**. Write or rewrite the file.
- **Execute**. Load the file into memory and execute it.
- **Append**. Write new information at the end of the file.
- **Delete**. Delete the file and free its space for possible reuse.
- **List**. List the name and attributes of the file.
- **Attribute change**. Changing the attributes of the file.

Other operations, such as renaming, copying, and editing the file, may also be controlled. For many systems, however, these higher-level functions may be implemented by a system program that makes lower-level system calls. Protection is provided at only the lower level. For instance, copying a file may be implemented simply by a sequence of read requests. In this case, a user with read access can also cause the file to be copied, printed, and so on.

Many protection mechanisms have been proposed. Each has advantages and disadvantages and must be appropriate for its intended application. A small computer system that is used by only a few members of a research group, for example, may not need the same types of protection as a large corporate computer that is used for research, finance, and personnel operations. We discuss some approaches to protection in the following sections and present a more complete treatment in Chapter 17.

### 13.4.2 Access Control

The most common approach to the protection problem is to make access dependent on the identity of the user. Different users may need different types of access to a file or directory. The most general scheme to implement identity-dependent access is to associate with each file and directory an **access-control list** (**ACL**) specifying user names and the types of access allowed for each user. When a user requests access to a particular file, the operating system checks the access list associated with that file. If that user is listed for the requested access, the access is allowed. Otherwise, a protection violation occurs, and the user job is denied access to the file.

This approach has the advantage of enabling complex access methodologies. The main problem with access lists is their length. If we want to allow everyone to read a file, we must list all users with read access. This technique has two undesirable consequences:

- Constructing such a list may be a tedious and unrewarding task, especially if we do not know in advance the list of users in the system.
- The directory entry, previously of fixed size, now must be of variable size, resulting in more complicated space management.

These problems can be resolved by use of a condensed version of the access list.

To condense the length of the access-control list, many systems recognize three classifications of users in connection with each file:

- **Owner**. The user who created the file is the owner.
- **Group**. A set of users who are sharing the file and need similar access is a group, or work group.
- **Other**. All other users in the system.

The most common recent approach is to combine access-control lists with the more general (and easier to implement) owner, group, and universe access-control scheme just described. For example, Solaris uses the three categories of access by default but allows access-control lists to be added to specific files and directories when more fine-grained access control is desired.

To illustrate, consider a person, Sara, who is writing a new book. She has hired three graduate students (Jim, Dawn, and Jill) to help with the project. The text of the book is kept in a file named `book.tex`. The protection associated with this file is as follows:

- Sara should be able to invoke all operations on the file.
- Jim, Dawn, and Jill should be able only to read and write the file; they should not be allowed to delete the file.
- All other users should be able to read, but not write, the file. (Sara is interested in letting as many people as possible read the text so that she can obtain feedback.)

### *PERMISSIONS IN A UNIX SYSTEM*

In the UNIX system, directory protection and file protection are handled similarly. Associated with each file and directory are three fields—owner, group, and universe—each consisting of the three bits `rwx`, where `r` controls **r**ead access, `w` controls **w**rite access, and `x` controls e**x**ecution. Thus, a user can list the content of a subdirectory only if the `r` bit is set in the appropriate field. Similarly, a user can change his current directory to another current directory (say, `foo`) only if the `x` bit associated with the `foo` subdirectory is set in the appropriate field.

A sample directory listing from a UNIX environment is shown in below:

```
-rw-rw-r--    1 pbg   staff      31200   Sep 3 08:30    intro.ps
drwx------    5 pbg   staff        512   Jul 8 09.33    private/
drwxrwxr-x    2 pbg   staff        512   Jul 8 09:35    doc/
drwxrwx---    2 jwg   student      512   Aug 3 14:13    student-proj/
-rw-r--r--    1 pbg   staff       9423   Feb 24 2017    program.c
-rwxr-xr-x    1 pbg   staff      20471   Feb 24 2017    program
drwx--x--x    4 tag   faculty      512   Jul 31 10:31   lib/
drwx------    3 pbg   staff       1024   Aug 29 06:52   mail/
drwxrwxrwx    3 pbg   staff        512   Jul 8 09:35    test/
```

The first field describes the protection of the file or directory. A `d` as the first character indicates a subdirectory. Also shown are the number of links to the file, the owner's name, the group's name, the size of the file in bytes, the date of last modification, and finally the file's name (with optional extension).

To achieve such protection, we must create a new group—say, `text`—with members Jim, Dawn, and Jill. The name of the group, `text`, must then be associated with the file `book.tex`, and the access rights must be set in accordance with the policy we have outlined.

Now consider a visitor to whom Sara would like to grant temporary access to Chapter 1. The visitor cannot be added to the `text` group because that would give him access to all chapters. Because a file can be in only one group, Sara cannot add another group to Chapter 1. With the addition of access-control-list functionality, though, the visitor can be added to the access control list of Chapter 1.

For this scheme to work properly, permissions and access lists must be controlled tightly. This control can be accomplished in several ways. For example, in the UNIX system, groups can be created and modified only by the manager of the facility (or by any superuser). Thus, control is achieved through human interaction. Access lists are discussed further in Section 17.6.2.

With the more limited protection classification, only three fields are needed to define protection. Often, each field is a collection of bits, and each bit either allows or prevents the access associated with it. For example, the UNIX system defines three fields of three bits each—`rwx`, where `r` controls read access, `w` controls write access, and `x` controls execution. A separate field is kept for the file owner, for the file's group, and for all other users. In this scheme, nine bits per file are needed to record protection information. Thus, for our example, the protection fields for the file `book.tex` are as follows: for the owner Sara, all bits are set; for the group `text`, the `r` and `w` bits are set; and for the universe, only the `r` bit is set.

One difficulty in combining approaches comes in the user interface. Users must be able to tell when the optional ACL permissions are set on a file. In the Solaris example, a "+" is appended to the regular permissions, as in:

```
19 -rw-r--r--+ 1 jim staff 130 May 25 22:13 file1
```

A separate set of commands, `setfacl` and `getfacl`, is used to manage the ACLs.

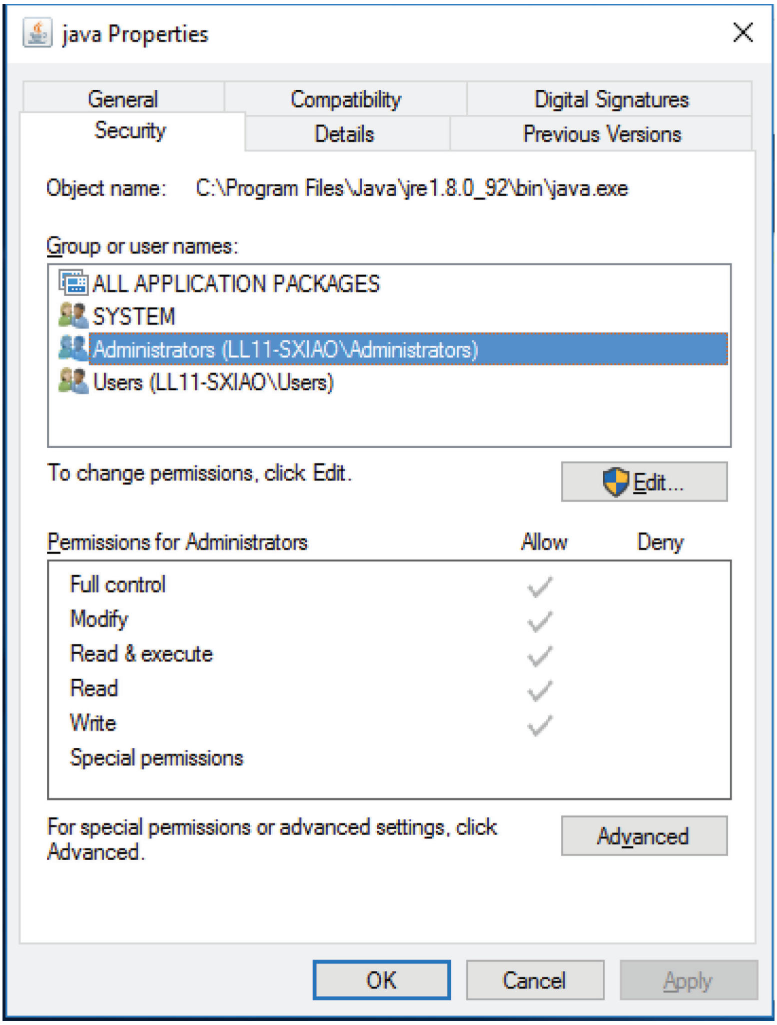
Windows users typically manage access-control lists via the GUI. Figure 13.12 shows a file-permission window on Windows 7 NTFS file system. In this example, user "guest" is specifically denied access to the file `ListPanel.java`.

Another difficulty is assigning precedence when permission and ACLs conflict. For example, if Walter is in a file's group, which has read permission, but the file has an ACL granting Walter read and write permission, should a write by Walter be granted or denied? Solaris and other operating systems give ACLs precedence (as they are more fine-grained and are not assigned by default). This follows the general rule that specificity should have priority.

### 13.4.3 Other Protection Approaches

Another approach to the protection problem is to associate a password with each file. Just as access to the computer system is often controlled by a password, access to each file can be controlled in the same way. If the passwords are chosen randomly and changed often, this scheme may be effective in limiting access to a file. The use of passwords has a few disadvantages, however. First, the number of passwords that a user needs to remember may become large, making the scheme impractical. Second, if only one password is used for all the files, then once it is discovered, all files are accessible; protection is on an all-or-none basis. Some systems allow a user to associate a password with a subdirectory, rather than with an individual file, to address this problem. More commonly encryption of a partition or individual files provides strong protection, but password management is key.

In a multilevel directory structure, we need to protect not only individual files but also collections of files in subdirectories; that is, we need to provide a mechanism for directory protection. The directory operations that must be protected are somewhat different from the file operations. We want to control the creation and deletion of files in a directory. In addition, we probably want to control whether a user can determine the existence of a file in a directory. Sometimes, knowledge of the existence and name of a file is significant in itself. Thus, listing the contents of a directory must be a protected operation. Similarly, if a path name refers to a file in a directory, the user must be allowed access to both the directory and the file. In systems where files may have numerous path names (such as acyclic and general graphs), a given user may have different access rights to a particular file, depending on the path name used.



Object name: C:\Program Files\Java\jre1.8.0_92\bin\java.exe

Group or user names:

- ALL APPLICATION PACKAGES
- SYSTEM
- Administrators (LL11-SXIAO\Administrators)
- Users (LL11-SXIAO\Users)

To change permissions, click Edit.

| Permissions for Administrators | Allow | Deny |
|---|---|---|
| Full control | ✓ | |
| Modify | ✓ | |
| Read & execute | ✓ | |
| Read | ✓ | |
| Write | ✓ | |
| Special permissions | | |

For special permissions or advanced settings, click Advanced.

**Figure 13.12** Windows 10 access-control list management.

## 13.5 Memory-Mapped Files

There is one other method of accessing files, and it is very commonly used. Consider a sequential read of a file on disk using the standard system calls `open()`, `read()`, and `write()`. Each file access requires a system call and disk access. Alternatively, we can use the virtual memory techniques discussed in Chapter 10 to treat file I/O as routine memory accesses. This approach, known as **memory mapping** a file, allows a part of the virtual address space to be logically associated with the file. As we shall see, this can lead to significant performance increases.

### 13.5.1 Basic Mechanism

Memory mapping a file is accomplished by mapping a disk block to a page (or pages) in memory. Initial access to the file proceeds through ordinary demand paging, resulting in a page fault. However, a page-sized portion of the file is read from the file system into a physical page (some systems may opt to read in more than a page-sized chunk of memory at a time). Subsequent reads and writes to the file are handled as routine memory accesses. Manipulating files through memory rather than incurring the overhead of using the `read()` and `write()` system calls simplifies and speeds up file access and usage.

Note that writes to the file mapped in memory are not necessarily immediate (synchronous) writes to the file on secondary storage. Generally, systems update the file based on changes to the memory image only when the file is closed. Under memory pressure, systems will have any intermediate changes to swap space to not lose them when freeing memory for other uses. When the file is closed, all the memory-mapped data are written back to the file on secondary storage and removed from the virtual memory of the process.

Some operating systems provide memory mapping only through a specific system call and use the standard system calls to perform all other file I/O. However, some systems choose to memory-map a file regardless of whether the file was specified as memory-mapped. Let's take Solaris as an example. If a file is specified as memory-mapped (using the `mmap()` system call), Solaris maps the file into the address space of the process. If a file is opened and accessed using ordinary system calls, such as `open()`, `read()`, and `write()`, Solaris still memory-maps the file; however, the file is mapped to the kernel address space. Regardless of how the file is opened, then, Solaris treats all file I/O as memory-mapped, allowing file access to take place via the efficient memory subsystem and avoiding system call overhead caused by each traditional `read()` and `write()`.

Multiple processes may be allowed to map the same file concurrently, to allow sharing of data. Writes by any of the processes modify the data in virtual memory and can be seen by all others that map the same section of the file. Given our earlier discussions of virtual memory, it should be clear how the sharing of memory-mapped sections of memory is implemented: the virtual memory map of each sharing process points to the same page of physical memory—the page that holds a copy of the disk block. This memory sharing is illustrated in Figure 13.13. The memory-mapping system calls can also support copy-on-write functionality, allowing processes to share a file in read-only mode but to have their own copies of any data they modify. So that access to the shared data is coordinated, the processes involved might use one of the mechanisms for achieving mutual exclusion described in Chapter 6.

Quite often, shared memory is in fact implemented by memory mapping files. Under this scenario, processes can communicate using shared memory by having the communicating processes memory-map the same file into their virtual address spaces. The memory-mapped file serves as the region of shared memory between the communicating processes (Figure 13.14). We have already seen this in Section 3.5, where a POSIX shared-memory object is created and each communicating process memory-maps the object into its address space. In the following section, we discuss support in the Windows API for shared memory using memory-mapped files.

### 13.5.2 Shared Memory in the Windows API

The general outline for creating a region of shared memory using memory-mapped files in the Windows API involves first creating a **file mapping** for the

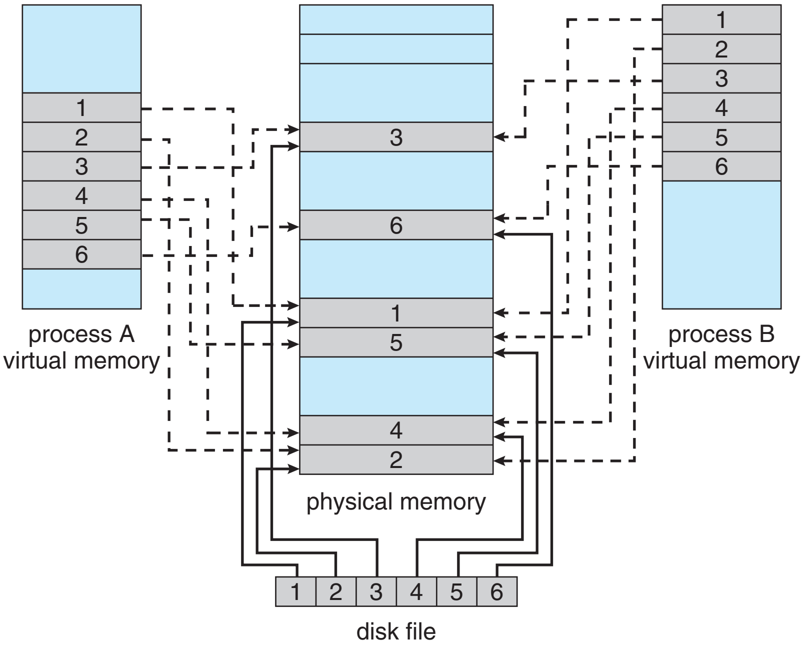

1
2
3
4
5
6
process A
virtual memory
3
6
1
5
4
2
physical memory
1
2
3
4
5
6
process B
virtual memory
1 2 3 4 5 6
disk file

**Figure 13.13** Memory-mapped files.

file to be mapped and then establishing a **view** of the mapped file in a process's virtual address space. A second process can then open and create a view of the mapped file in its virtual address space. The mapped file represents the shared-memory object that will enable communication to take place between the processes.

We next illustrate these steps in more detail. In this example, a producer process first creates a shared-memory object using the memory-mapping features available in the Windows API. The producer then writes a message to shared memory. After that, a consumer process opens a mapping to the shared-memory object and reads the message written by the consumer.

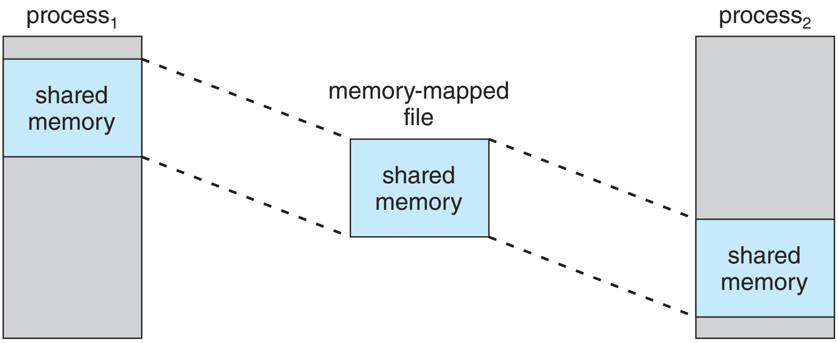

process₁
shared
memory
memory-mapped
file
shared
memory
process₂
shared
memory

**Figure 13.14** Shared memory using memory-mapped I/O.

To establish a memory-mapped file, a process first opens the file to be mapped with the `CreateFile()` function, which returns a `HANDLE` to the opened file. The process then creates a mapping of this file `HANDLE` using the `CreateFileMapping()` function. Once the file mapping is done, the process establishes a view of the mapped file in its virtual address space with the `MapViewOfFile()` function. The view of the mapped file represents the portion of the file being mapped in the virtual address space of the process—the entire file or only a portion of it may be mapped. This sequence in the program

```
#include <windows.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
   HANDLE hFile, hMapFile;
   LPVOID lpMapAddress;

   hFile = CreateFile("temp.txt", /* file name */
      GENERIC_READ | GENERIC_WRITE, /* read/write access */
      0, /* no sharing of the file */
      NULL, /* default security */
      OPEN_ALWAYS, /* open new or existing file */
      FILE_ATTRIBUTE_NORMAL, /* routine file attributes */
      NULL); /* no file template */

   hMapFile = CreateFileMapping(hFile, /* file handle */
      NULL, /* default security */
      PAGE_READWRITE, /* read/write access to mapped pages */
      0, /* map entire file */
      0,
      TEXT("SharedObject")); /* named shared memory object */

   lpMapAddress = MapViewOfFile(hMapFile, /* mapped object handle */
      FILE_MAP_ALL_ACCESS, /* read/write access */
      0, /* mapped view of entire file */
      0,
      0);

   /* write to shared memory */
   sprintf(lpMapAddress,"Shared memory message");

   UnmapViewOfFile(lpMapAddress);
   CloseHandle(hFile);
   CloseHandle(hMapFile);
}
```

**Figure 13.15** Producer writing to shared memory using the Windows API.

is shown in Figure 13.15. (We eliminate much of the error checking for code brevity.)

The call to `CreateFileMapping()` creates a **named shared-memory object** called `SharedObject`. The consumer process will communicate using this shared-memory segment by creating a mapping to the same named object. The producer then creates a view of the memory-mapped file in its virtual address space. By passing the last three parameters the value 0, it indicates that the mapped view is the entire file. It could instead have passed values specifying an offset and size, thus creating a view containing only a subsection of the file. (It is important to note that the entire mapping may not be loaded into memory when the mapping is established. Rather, the mapped file may be demand-paged, thus bringing pages into memory only as they are accessed.) The `MapViewOfFile()` function returns a pointer to the shared-memory object; any accesses to this memory location are thus accesses to the memory-mapped file. In this instance, the producer process writes the message "`Shared memory message`" to shared memory.

A program illustrating how the consumer process establishes a view of the named shared-memory object is shown in Figure 13.16. This program is

```
#include <windows.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
   HANDLE hMapFile;
   LPVOID lpMapAddress;

   hMapFile = OpenFileMapping(FILE_MAP_ALL_ACCESS, /* R/W access */
      FALSE, /* no inheritance */
      TEXT("SharedObject")); /* name of mapped file object */

   lpMapAddress = MapViewOfFile(hMapFile, /* mapped object handle */
      FILE_MAP_ALL_ACCESS, /* read/write access */
      0, /* mapped view of entire file */
      0,
      0);

   /* read from shared memory */
   printf("Read message %s", lpMapAddress);

   UnmapViewOfFile(lpMapAddress);
   CloseHandle(hMapFile);
}
```

**Figure 13.16** Consumer reading from shared memory using the Windows API.

somewhat simpler than the one shown in Figure 13.15, as all that is necessary is for the process to create a mapping to the existing named shared-memory object. The consumer process must also create a view of the mapped file, just as the producer process did in the program in Figure 13.15. The consumer then reads from shared memory the message "`Shared memory message`" that was written by the producer process.

Finally, both processes remove the view of the mapped file with a call to `UnmapViewOfFile()`. We provide a programming exercise at the end of this chapter using shared memory with memory mapping in the Windows API.

## 13.6 Summary

- A file is an abstract data type defined and implemented by the operating system. It is a sequence of logical records. A logical record may be a byte, a line (of fixed or variable length), or a more complex data item. The operating system may specifically support various record types or may leave that support to the application program.
- A major task for the operating system is to map the logical file concept onto physical storage devices such as hard disk or NVM device. Since the physical record size of the device may not be the same as the logical record size, it may be necessary to order logical records into physical records. Again, this task may be supported by the operating system or left for the application program.
- Within a file system, it is useful to create directories to allow files to be organized. A single-level directory in a multiuser system causes naming problems, since each file must have a unique name. A two-level directory solves this problem by creating a separate directory for each user's files. The directory lists the files by name and includes the file's location on the disk, length, type, owner, time of creation, time of last use, and so on.
- The natural generalization of a two-level directory is a tree-structured directory. A tree-structured directory allows a user to create subdirectories to organize files. Acyclic-graph directory structures enable users to share subdirectories and files but complicate searching and deletion. A general graph structure allows complete flexibility in the sharing of files and directories but sometimes requires garbage collection to recover unused disk space.
- Remote file systems present challenges in reliability, performance, and security. Distributed information systems maintain user, host, and access information so that clients and servers can share state information to manage use and access.
- Since files are the main information-storage mechanism in most computer systems, file protection is needed on multiuser systems. Access to files can be controlled separately for each type of access—read, write, execute, append, delete, list directory, and so on. File protection can be provided by access lists, passwords, or other techniques.

## Practice Exercises

**13.1** Some systems automatically delete all user files when a user logs off or a job terminates, unless the user explicitly requests that they be kept. Other systems keep all files unless the user explicitly deletes them. Discuss the relative merits of each approach.

**13.2** Why do some systems keep track of the type of a file, while still others leave it to the user and others simply do not implement multiple file types? Which system is "better"?

**13.3** Similarly, some systems support many types of structures for a file's data, while others simply support a stream of bytes. What are the advantages and disadvantages of each approach?

**13.4** Could you simulate a multilevel directory structure with a single-level directory structure in which arbitrarily long names can be used? If your answer is yes, explain how you can do so, and contrast this scheme with the multilevel directory scheme. If your answer is no, explain what prevents your simulation's success. How would your answer change if file names were limited to seven characters?

**13.5** Explain the purpose of the `open()` and `close()` operations.

**13.6** In some systems, a subdirectory can be read and written by an authorized user, just as ordinary files can be.

a. Describe the protection problems that could arise.

b. Suggest a scheme for dealing with each of these protection problems.

**13.7** Consider a system that supports 5,000 users. Suppose that you want to allow 4,990 of these users to be able to access one file.

a. How would you specify this protection scheme in UNIX?

b. Can you suggest another protection scheme that can be used more effectively for this purpose than the scheme provided by UNIX?

**13.8** Researchers have suggested that, instead of having an access-control list associated with each file (specifying which users can access the file, and how), we should have a **user control list** associated with each user (specifying which files a user can access, and how). Discuss the relative merits of these two schemes.

## Further Reading

A multilevel directory structure was first implemented on the MULTICS system ([Organick (1972)]). Most operating systems now implement multilevel directory structures. These include Linux ([Love (2010)]), macOS ([Singh (2007)]), Solaris ([McDougall and Mauro (2007)]), and all versions of Windows ([Russinovich et al. (2017)]).

A general discussion of Solaris file systems is found in the Sun *System Administration Guide: Devices and File Systems* (http://docs.sun.com/app/docs/doc/817-5093).

The network file system (NFS), designed by Sun Microsystems, allows directory structures to be spread across networked computer systems. NFS Version 4 is described in RFC3505 (http://www.ietf.org/rfc/rfc3530.txt).

A great source of the meanings of computer jargon is http://www.catb.org/esr/jargon/.

## Bibliography

**[Love (2010)]** R. Love, *Linux Kernel Development*, Third Edition, Developer's Library (2010).

**[McDougall and Mauro (2007)]** R. McDougall and J. Mauro, *Solaris Internals,* Second Edition, Prentice Hall (2007).

**[Organick (1972)]** E. I. Organick, *The Multics System: An Examination of Its Structure*, MIT Press (1972).

**[Russinovich et al. (2017)]** M. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals–Part 1,* Seventh Edition, Microsoft Press (2017).

**[Singh (2007)]** A. Singh, *Mac OS X Internals: A Systems Approach*, Addison-Wesley (2007).

## Chapter 13 Exercises

**13.9** Consider a file system in which a file can be deleted and its disk space reclaimed while links to that file still exist. What problems may occur if a new file is created in the same storage area or with the same absolute path name? How can these problems be avoided?

**13.10** The open-file table is used to maintain information about files that are currently open. Should the operating system maintain a separate table for each user or maintain just one table that contains references to files that are currently being accessed by all users? If the same file is being accessed by two different programs or users, should there be separate entries in the open-file table? Explain.

**13.11** What are the advantages and disadvantages of providing mandatory locks instead of advisory locks whose use is left to users' discretion?

**13.12** Provide examples of applications that typically access files according to the following methods:

- Sequential
- Random

**13.13** Some systems automatically open a file when it is referenced for the first time and close the file when the job terminates. Discuss the advantages and disadvantages of this scheme compared with the more traditional one, where the user has to open and close the file explicitly.

**13.14** If the operating system knew that a certain application was going to access file data in a sequential manner, how could it exploit this information to improve performance?

**13.15** Give an example of an application that could benefit from operating-system support for random access to indexed files.

**13.16** Some systems provide file sharing by maintaining a single copy of a file. Other systems maintain several copies, one for each of the users sharing the file. Discuss the relative merits of each approach.

CHAPTER 14

# File-System Implementation

As we saw in Chapter 13, the file system provides the mechanism for on-line storage and access to file contents, including data and programs. File systems usually reside permanently on secondary storage, which is designed to hold a large amount of data. This chapter is primarily concerned with issues surrounding file storage and access on the most common secondary-storage media, hard disk drives and nonvolatile memory devices. We explore ways to structure file use, to allocate storage space, to recover freed space, to track the locations of data, and to interface other parts of the operating system to secondary storage. Performance issues are considered throughout the chapter.

A given general-purpose operating system provides several file systems. Additionally, many operating systems allow administrators or users to add file systems. Why so many? File systems vary in many respects, including features, performance, reliability, and design goals, and different file systems may serve different purposes. For example, a temporary file system is used for fast storage and retrieval of nonpersistent files, while the default secondary storage file system (such as Linux ext4) sacrifices performance for reliability and features. As we've seen throughout this study of operating systems, there are plenty of choices and variations, making thorough coverage a challenge. In this chapter, we concentrate on the common denominators.

## CHAPTER OBJECTIVES

- Describe the details of implementing local file systems and directory structures.
- Discuss block allocation and free-block algorithms and trade-offs.
- Explore file system efficiency and performance issues.
- Look at recovery from file system failures.
- Describe the WAFL file system as a concrete example.

## 14.1 File-System Structure

Disks provide most of the secondary storage on which file systems are maintained. Two characteristics make them convenient for this purpose:

1. A disk can be rewritten in place; it is possible to read a block from the disk, modify the block, and write it back into the same block.
2. A disk can access directly any block of information it contains. Thus, it is simple to access any file either sequentially or randomly, and switching from one file to another requires the drive moving the read–write heads and waiting for the media to rotate.

Nonvolatile memory (NVM) devices are increasingly used for file storage and thus as a location for file systems. They differ from hard disks in that they cannot be rewritten in place and they have different performance characteristics. We discuss disk and NVM-device structure in detail in Chapter 11.

To improve I/O efficiency, I/O transfers between memory and mass storage are performed in units of **blocks**. Each block on a hard disk drive has one or more sectors. Depending on the disk drive, sector size is usually 512 bytes or 4,096 bytes. NVM devices usually have blocks of 4,096 bytes, and the transfer methods used are similar to those used by disk drives.

**File systems** provide efficient and convenient access to the storage device by allowing data to be stored, located, and retrieved easily. A file system poses two quite different design problems. The first problem is defining how the file system should look to the user. This task involves defining a file and its attributes, the operations allowed on a file, and the directory structure for organizing files. The second problem is creating algorithms and data structures to map the logical file system onto the physical secondary-storage devices.

The file system itself is generally composed of many different levels. The structure shown in Figure 14.1 is an example of a layered design. Each level in the design uses the features of lower levels to create new features for use by higher levels.

The **I/O control** level consists of device drivers and interrupt handlers to transfer information between the main memory and the disk system. A device driver can be thought of as a translator. Its input consists of high-level commands, such as “retrieve block 123.” Its output consists of low-level, hardware-specific instructions that are used by the hardware controller, which interfaces the I/O device to the rest of the system. The device driver usually writes specific bit patterns to special locations in the I/O controller’s memory to tell the controller which device location to act on and what actions to take. The details of device drivers and the I/O infrastructure are covered in Chapter 12.

The **basic file system** (called the “block I/O subsystem” in Linux) needs only to issue generic commands to the appropriate device driver to read and write blocks on the storage device. It issues commands to the drive based on logical block addresses. It is also concerned with I/O request scheduling. This layer also manages the memory buffers and caches that hold various file-system, directory, and data blocks. A block in the buffer is allocated before the transfer of a mass storage block can occur. When the buffer is full, the buffer manager must find more buffer memory or free up buffer space to allow a

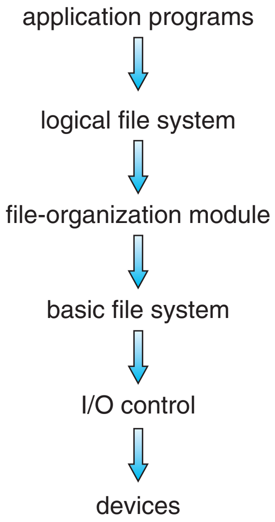


**Figure 14.1** Layered file system.

requested I/O to complete. Caches are used to hold frequently used file-system metadata to improve performance, so managing their contents is critical for optimum system performance.

The **file-organization module** knows about files and their logical blocks. Each file's logical blocks are numbered from 0 (or 1) through *N*. The file-organization module also includes the free-space manager, which tracks unallocated blocks and provides these blocks to the file-organization module when requested.

Finally, the **logical file system** manages metadata information. Metadata includes all of the file-system structure except the actual data (or contents of the files). The logical file system manages the directory structure to provide the file-organization module with the information the latter needs, given a symbolic file name. It maintains file structure via file-control blocks. A **file control block** (**FCB**) (an **inode** in UNIX file systems) contains information about the file, including ownership, permissions, and location of the file contents. The logical file system is also responsible for protection, as discussed in Chapters 13 and 17.

When a layered structure is used for file-system implementation, duplication of code is minimized. The I/O control and sometimes the basic file-system code can be used by multiple file systems. Each file system can then have its own logical file-system and file-organization modules. Unfortunately, layering can introduce more operating-system overhead, which may result in decreased performance. The use of layering, including the decision about how many layers to use and what each layer should do, is a major challenge in designing new systems.

Many file systems are in use today, and most operating systems support more than one. For example, most CD-ROMs are written in the ISO 9660 format, a standard format agreed on by CD-ROM manufacturers. In addition to removable-media file systems, each operating system has one or more disk-based file systems. UNIX uses the **UNIX file system** (**UFS**), which is based on the

Berkeley Fast File System (FFS). Windows supports disk file-system formats of FAT, FAT32, and NTFS (or Windows NT File System), as well as CD-ROM and DVD file-system formats. Although Linux supports over 130 different file systems, the standard Linux file system is known as the **extended file system**, with the most common versions being ext3 and ext4. There are also distributed file systems in which a file system on a server is mounted by one or more client computers across a network.

File-system research continues to be an active area of operating-system design and implementation. Google created its own file system to meet the company's specific storage and retrieval needs, which include high-performance access from many clients across a very large number of disks. Another interesting project is the FUSE file system, which provides flexibility in file-system development and use by implementing and executing file systems as user-level rather than kernel-level code. Using FUSE, a user can add a new file system to a variety of operating systems and can use that file system to manage her files.

## 14.2 File-System Operations

As was described in Section 13.1.2, operating systems implement `open()` and `close()` systems calls for processes to request access to file contents. In this section, we delve into the structures and operations used to implement file-system operations.

### 14.2.1 Overview

Several on-storage and in-memory structures are used to implement a file system. These structures vary depending on the operating system and the file system, but some general principles apply.

On storage, the file system may contain information about how to boot an operating system stored there, the total number of blocks, the number and location of free blocks, the directory structure, and individual files. Many of these structures are detailed throughout the remainder of this chapter. Here, we describe them briefly:

- A **boot control block** (per volume) can contain information needed by the system to boot an operating system from that volume. If the disk does not contain an operating system, this block can be empty. It is typically the first block of a volume. In UFS, it is called the **boot block**. In NTFS, it is the **partition boot sector**.
- A **volume control block** (per volume) contains volume details, such as the number of blocks in the volume, the size of the blocks, a free-block count and free-block pointers, and a free-FCB count and FCB pointers. In UFS, this is called a **superblock**. In NTFS, it is stored in the **master file table**.
- A directory structure (per file system) is used to organize the files. In UFS, this includes file names and associated inode numbers. In NTFS, it is stored in the master file table.

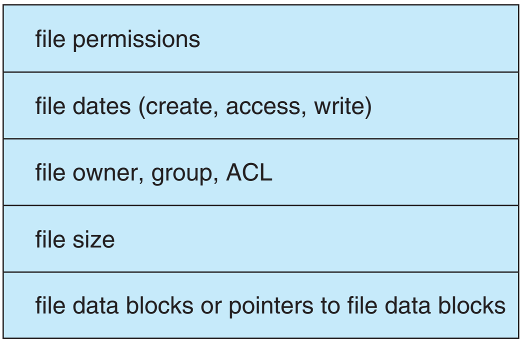
- A per-file FCB contains many details about the file. It has a unique identifier number to allow association with a directory entry. In NTFS, this information is actually stored within the master file table, which uses a relational database structure, with a row per file.

The in-memory information is used for both file-system management and performance improvement via caching. The data are loaded at mount time, updated during file-system operations, and discarded at dismount. Several types of structures may be included.

- An in-memory **mount table** contains information about each mounted volume.
- An in-memory directory-structure cache holds the directory information of recently accessed directories. (For directories at which volumes are mounted, it can contain a pointer to the volume table.)
- The **system-wide open-file table** contains a copy of the FCB of each open file, as well as other information.
- The **per-process open-file table** contains pointers to the appropriate entries in the system-wide open-file table, as well as other information, for all files the process has open.
- Buffers hold file-system blocks when they are being read from or written to a file system.

To create a new file, a process calls the logical file system. The logical file system knows the format of the directory structures. To create a new file, it allocates a new FCB. (Alternatively, if the file-system implementation creates all FCBs at file-system creation time, an FCB is allocated from the set of free FCBs.) The system then reads the appropriate directory into memory, updates it with the new file name and FCB, and writes it back to the file system. A typical FCB is shown in Figure 14.2.

Some operating systems, including UNIX, treat a directory exactly the same as a file—one with a "type" field indicating that it is a directory. Other oper-



file permissions

file dates (create, access, write)

file owner, group, ACL

file size

file data blocks or pointers to file data blocks

**Figure 14.2** A typical file-control block.

ating systems, including Windows, implement separate system calls for files and directories and treat directories as entities separate from files. Whatever the larger structural issues, the logical file system can call the file-organization module to map the directory I/O into storage block locations, which are passed on to the basic file system and I/O control system.

### 14.2.2 Usage

Now that a file has been created, it can be used for I/O. First, though, it must be opened. The `open()` call passes a file name to the logical file system. The `open()` system call first searches the system-wide open-file table to see if the file is already in use by another process. If it is, a per-process open-file table entry is created pointing to the existing system-wide open-file table. This algorithm can save substantial overhead. If the file is not already open, the directory structure is searched for the given file name. Parts of the directory structure are usually cached in memory to speed directory operations. Once the file is found, the FCB is copied into a system-wide open-file table in memory. This table not only stores the FCB but also tracks the number of processes that have the file open.

Next, an entry is made in the per-process open-file table, with a pointer to the entry in the system-wide open-file table and some other fields. These other fields may include a pointer to the current location in the file (for the next `read()` or `write()` operation) and the access mode in which the file is open. The `open()` call returns a pointer to the appropriate entry in the per-process file-system table. All file operations are then performed via this pointer. The file name may not be part of the open-file table, as the system has no use for it once the appropriate FCB is located on disk. It could be cached, though, to save time on subsequent opens of the same file. The name given to the entry varies. UNIX systems refer to it as a **file descriptor**; Windows refers to it as a **file handle**.

When a process closes the file, the per-process table entry is removed, and the system-wide entry's open count is decremented. When all users that have opened the file close it, any updated metadata are copied back to the disk-based directory structure, and the system-wide open-file table entry is removed.

The caching aspects of file-system structures should not be overlooked. Most systems keep all information about an open file, except for its actual data blocks, in memory. The BSD UNIX system is typical in its use of caches wherever disk I/O can be saved. Its average cache hit rate of 85 percent shows that these techniques are well worth implementing. The BSD UNIX system is described fully in Appendix C.

The operating structures of a file-system implementation are summarized in Figure 14.3.

## 14.3 Directory Implementation

The selection of directory-allocation and directory-management algorithms significantly affects the efficiency, performance, and reliability of the file system. In this section, we discuss the trade-offs involved in choosing one of these algorithms.

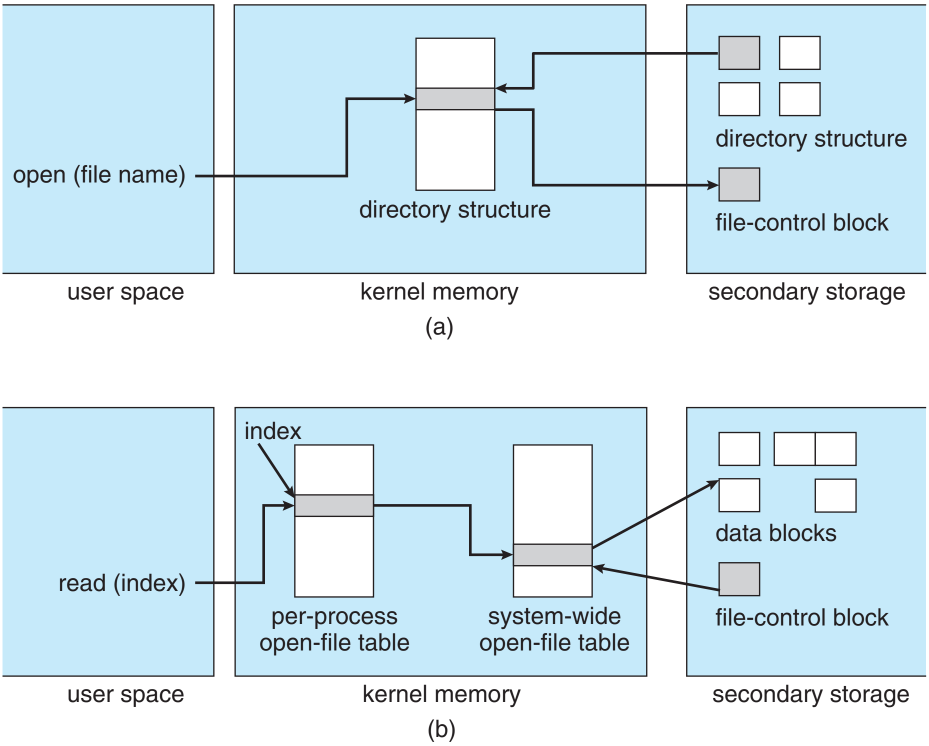

open (file name)
directory structure
directory structure
file-control block
user space
kernel memory
secondary storage
(a)
index
read (index)
per-process
open-file table
system-wide
open-file table
data blocks
file-control block
user space
kernel memory
secondary storage
(b)

**Figure 14.3** In-memory file-system structures. (a) File open. (b) File read.

### 14.3.1 Linear List

The simplest method of implementing a directory is to use a linear list of file names with pointers to the data blocks. This method is simple to program but time-consuming to execute. To create a new file, we must first search the directory to be sure that no existing file has the same name. Then, we add a new entry at the end of the directory. To delete a file, we search the directory for the named file and then release the space allocated to it. To reuse the directory entry, we can do one of several things. We can mark the entry as unused (by assigning it a special name, such as an all-blank name, assigning it an invalid inode number (such as 0), or by including a used–unused bit in each entry), or we can attach it to a list of free directory entries. A third alternative is to copy the last entry in the directory into the freed location and to decrease the length of the directory. A linked list can also be used to decrease the time required to delete a file.

The real disadvantage of a linear list of directory entries is that finding a file requires a linear search. Directory information is used frequently, and users will notice if access to it is slow. In fact, many operating systems implement a software cache to store the most recently used directory information. A cache hit avoids the need to constantly reread the information from secondary storage. A sorted list allows a binary search and decreases the average search time. However, the requirement that the list be kept sorted may complicate creating and deleting files, since we may have to move substantial amounts of directory information to maintain a sorted directory. A more sophisticated tree data structure, such as a balanced tree, might help here. An advantage of the sorted list is that a sorted directory listing can be produced without a separate sort step.

### 14.3.2 Hash Table

Another data structure used for a file directory is a hash table. Here, a linear list stores the directory entries, but a hash data structure is also used. The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list. Therefore, it can greatly decrease the directory search time. Insertion and deletion are also fairly straightforward, although some provision must be made for collisions—situations in which two file names hash to the same location.

The major difficulties with a hash table are its generally fixed size and the dependence of the hash function on that size. For example, assume that we make a linear-probing hash table that holds 64 entries. The hash function converts file names into integers from 0 to 63 (for instance, by using the remainder of a division by 64). If we later try to create a 65th file, we must enlarge the directory hash table—say, to 128 entries. As a result, we need a new hash function that must map file names to the range 0 to 127, and we must reorganize the existing directory entries to reflect their new hash-function values.

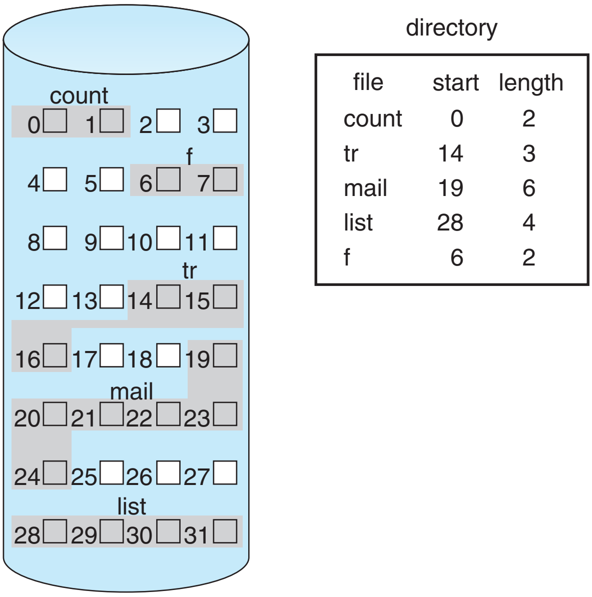
Alternatively, we can use a chained-overflow hash table. Each hash entry can be a linked list instead of an individual value, and we can resolve collisions by adding the new entry to the linked list. Lookups may be somewhat slowed, because searching for a name might require stepping through a linked list of colliding table entries. Still, this method is likely to be much faster than a linear search through the entire directory.

## 14.4 Allocation Methods

The direct-access nature of secondary storage gives us flexibility in the implementation of files. In almost every case, many files are stored on the same device. The main problem is how to allocate space to these files so that storage space is utilized effectively and files can be accessed quickly. Three major methods of allocating secondary storage space are in wide use: contiguous, linked, and indexed. Each method has advantages and disadvantages. Although some systems support all three, it is more common for a system to use one method for all files within a file-system type.

### 14.4.1 Contiguous Allocation

**Contiguous allocation** requires that each file occupy a set of contiguous blocks on the device. Device addresses define a linear ordering on the device. With this ordering, assuming that only one job is accessing the device, accessing block $b + 1$ after block $b$ normally requires no head movement. When head movement is needed (from the last sector of one cylinder to the first sector of the next cylinder), the head need only move from one track to the next. Thus, for HDDs, the number of disk seeks required for accessing contiguously allocated files is



directory

| file | start | length |
|---|---|---|
| count | 0 | 2 |
| tr | 14 | 3 |
| mail | 19 | 6 |
| list | 28 | 4 |
| f | 6 | 2 |

**Figure 14.4** Contiguous allocation of disk space.

minimal (assuming blocks with close logical addresses are close physically), as is seek time when a seek is finally needed.

Contiguous allocation of a file is defined by the address of the first block and length (in block units) of the file. If the file is $n$ blocks long and starts at location $b$, then it occupies blocks $b, b + 1, b + 2, ..., b + n - 1$. The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file (Figure 14.4). Contiguous allocation is easy to implement but has limitations, and is therefore not used in modern file systems.

Accessing a file that has been allocated contiguously is easy. For sequential access, the file system remembers the address of the last block referenced and, when necessary, reads the next block. For direct access to block $i$ of a file that starts at block $b$, we can immediately access block $b + i$. Thus, both sequential and direct access can be supported by contiguous allocation.

Contiguous allocation has some problems, however. One difficulty is finding space for a new file. The system chosen to manage free space determines how this task is accomplished; these management systems are discussed in Section 14.5. Any management system can be used, but some are slower than others.

The contiguous-allocation problem can be seen as a particular application of the general **dynamic storage-allocation** problem discussed in Section 9.2, which involves how to satisfy a request of size $n$ from a list of free holes. First fit and best fit are the most common strategies used to select a free hole from the set of available holes. Simulations have shown that both first fit and best fit are more efficient than worst fit in terms of both time and storage utilization. Neither first fit nor best fit is clearly best in terms of storage utilization, but first fit is generally faster.

All these algorithms suffer from the problem of **external fragmentation**. As files are allocated and deleted, the free storage space is broken into little pieces.

External fragmentation exists whenever free space is broken into chunks. It becomes a problem when the largest contiguous chunk is insufficient for a request; storage is fragmented into a number of holes, none of which is large enough to store the data. Depending on the total amount of disk storage and the average file size, external fragmentation may be a minor or a major problem.

One strategy for preventing loss of significant amounts of storage space to external fragmentation is to copy an entire file system onto another device. The original device is then freed completely, creating one large contiguous free space. We then copy the files back onto the original device by allocating contiguous space from this one large hole. This scheme effectively **compacts** all free space into one contiguous space, solving the fragmentation problem. The cost of this compaction is time, however, and the cost can be particularly high for large storage devices. Compacting these devices may take hours and may be necessary on a weekly basis. Some systems require that this function be done **off-line**, with the file system unmounted. During this **down time**, normal system operation generally cannot be permitted, so such compaction is avoided at all costs on production machines. Most modern systems that need defragmentation can perform it **on-line** during normal system operations, but the performance penalty can be substantial.

Another problem with contiguous allocation is determining how much space is needed for a file. When the file is created, the total amount of space it will need must be found and allocated. How does the creator (program or person) know the size of the file to be created? In some cases, this determination may be fairly simple (copying an existing file, for example). In general, however, the size of an output file may be difficult to estimate.

If we allocate too little space to a file, we may find that the file cannot be extended. Especially with a best-fit allocation strategy, the space on both sides of the file may be in use. Hence, we cannot make the file larger in place. Two possibilities then exist. First, the user program can be terminated, with an appropriate error message. The user must then allocate more space and run the program again. These repeated runs may be costly. To prevent them, the user will normally overestimate the amount of space needed, resulting in considerable wasted space. The other possibility is to find a larger hole, copy the contents of the file to the new space, and release the previous space. This series of actions can be repeated as long as space exists, although it can be time consuming. The user need never be informed explicitly about what is happening, however; the system continues despite the problem, although more and more slowly.

Even if the total amount of space needed for a file is known in advance, preallocation may be inefficient. A file that will grow slowly over a long period (months or years) must be allocated enough space for its final size, even though much of that space will be unused for a long time. The file therefore has a large amount of internal fragmentation.

To minimize these drawbacks, an operating system can use a modified contiguous-allocation scheme. Here, a contiguous chunk of space is allocated initially. Then, if that amount proves not to be large enough, another chunk of contiguous space, known as an **extent**, is added. The location of a file's blocks is then recorded as a location and a block count, plus a link to the first block of the next extent. On some systems, the owner of the file can set the extent size, but this setting results in inefficiencies if the owner is incorrect. Internal fragmentation can still be a problem if the extents are too large, and external fragmentation can become a problem as extents of varying sizes are allocated and deallocated. The commercial Symantec Veritas file system uses extents to optimize performance. Veritas is a high-performance replacement for the standard UNIX UFS.

### 14.4.2 Linked Allocation

**Linked allocation** solves all problems of contiguous allocation. With linked allocation, each file is a linked list of storage blocks; the blocks may be scattered anywhere on the device. The directory contains a pointer to the first and last blocks of the file. For example, a file of five blocks might start at block 9 and continue at block 16, then block 1, then block 10, and finally block 25 (Figure 14.5). Each block contains a pointer to the next block. These pointers are not made available to the user. Thus, if each block is 512 bytes in size, and a block address (the pointer) requires 4 bytes, then the user sees blocks of 508 bytes.

To create a new file, we simply create a new entry in the directory. With linked allocation, each directory entry has a pointer to the first block of the file. This pointer is initialized to `null` (the end-of-list pointer value) to signify an empty file. The size field is also set to 0. A write to the file causes the free-space management system to find a free block, and this new block is written to and is linked to the end of the file. To read a file, we simply read blocks by following the pointers from block to block. There is no external fragmentation with linked allocation, and any free block on the free-space list can be used to satisfy a request. The size of a file need not be declared when the file is created. A file can continue to grow as long as free blocks are available. Consequently, it is never necessary to compact disk space.

Linked allocation does have disadvantages, however. The major problem is that it can be used effectively only for sequential-access files. To find the *i*th

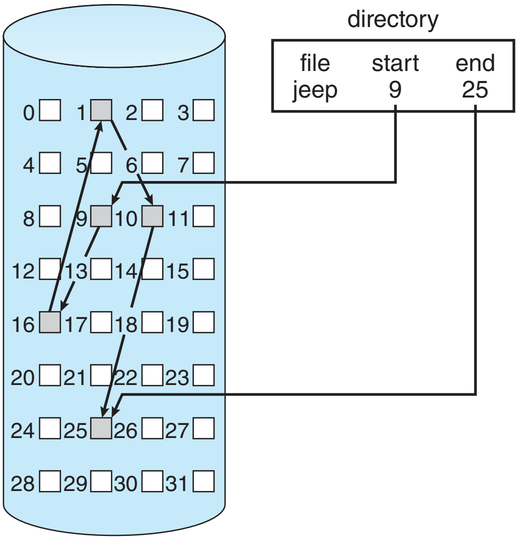


**Figure 14.5** Linked allocation of disk space.

block of a file, we must start at the beginning of that file and follow the pointers until we get to the *i*th block. Each access to a pointer requires a storage device read, and some require an HDD seek. Consequently, it is inefficient to support a direct-access capability for linked-allocation files.

Another disadvantage is the space required for the pointers. If a pointer requires 4 bytes out of a 512-byte block, then 0.78 percent of the disk is being used for pointers, rather than for information. Each file requires slightly more space than it would otherwise.

The usual solution to this problem is to collect blocks into multiples, called **clusters**, and to allocate clusters rather than blocks. For instance, the file system may define a cluster as four blocks and operate on the secondary storage device only in cluster units. Pointers then use a much smaller percentage of the file's space. This method allows the logical-to-physical block mapping to remain simple but improves HDD throughput (because fewer disk-head seeks are required) and decreases the space needed for block allocation and free-list management. The cost of this approach is an increase in internal fragmentation, because more space is wasted when a cluster is partially full than when a block is partially full. Also random I/O performance suffers because a request for a small amount of data transfers a large amount of data. Clusters can be used to improve the disk-access time for many other algorithms as well, so they are used in most file systems.

Yet another problem of linked allocation is reliability. Recall that the files are linked together by pointers scattered all over the device, and consider what would happen if a pointer was lost or damaged. A bug in the operating-system software or a hardware failure might result in picking up the wrong pointer. This error could in turn result in linking into the free-space list or into another file. One partial solution is to use doubly linked lists, and another is to store the file name and relative block number in each block. However, these schemes require even more overhead for each file.

An important variation on linked allocation is the use of a **file-allocatio table** (**FAT**). This simple but efficient method of disk-space allocation was used by the MS-DOS operating system. A section of storage at the beginning of each volume is set aside to contain the table. The table has one entry for each block and is indexed by block number. The FAT is used in much the same way as a linked list. The directory entry contains the block number of the first block of the file. The table entry indexed by that block number contains the block number of the next block in the file. This chain continues until it reaches the last block, which has a special end-of-file value as the table entry. An unused block is indicated by a table value of 0. Allocating a new block to a file is a simple matter of finding the first 0-valued table entry and replacing the previous end-of-file value with the address of the new block. The 0 is then replaced with the end-of-file value. An illustrative example is the FAT structure shown in Figure 14.6 for a file consisting of disk blocks 217, 618, and 339.

The FAT allocation scheme can result in a significant number of disk head seeks, unless the FAT is cached. The disk head must move to the start of the volume to read the FAT and find the location of the block in question, then move to the location of the block itself. In the worst case, both moves occur for each of the blocks. A benefit is that random-access time is improved, because the disk head can find the location of any block by reading the information in the FAT.

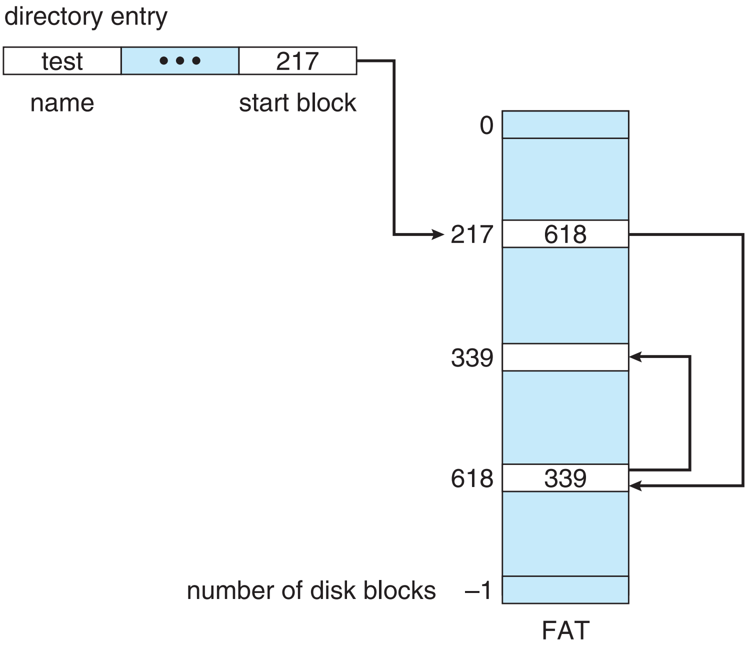


**Figure 14.6** File-allocation table.

### 14.4.3 Indexed Allocation

Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation. However, in the absence of a FAT, linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered with the blocks themselves all over the disk and must be retrieved in order. **Indexed allocation** solves this problem by bringing all the pointers together into one location: the **index block**.

Each file has its own index block, which is an array of storage-block addresses. The *i*th entry in the index block points to the *i*th block of the file. The directory contains the address of the index block (Figure 14.7). To find and read the *i*th block, we use the pointer in the *i*th index-block entry. This scheme is similar to the paging scheme described in Section 9.3.

When the file is created, all pointers in the index block are set to `null`. When the *i*th block is first written, a block is obtained from the free-space manager, and its address is put in the *i*th index-block entry.

Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the storage device can satisfy a request for more space. Indexed allocation does suffer from wasted space, however. The pointer overhead of the index block is generally greater than the pointer overhead of linked allocation. Consider a common case in which we have a file of only one or two blocks. With linked allocation, we lose the space of only one pointer per block. With indexed allocation, an entire index block must be allocated, even if only one or two pointers will be non-`null`.

This point raises the question of how large the index block should be. Every file must have an index block, so we want the index block to be as small as

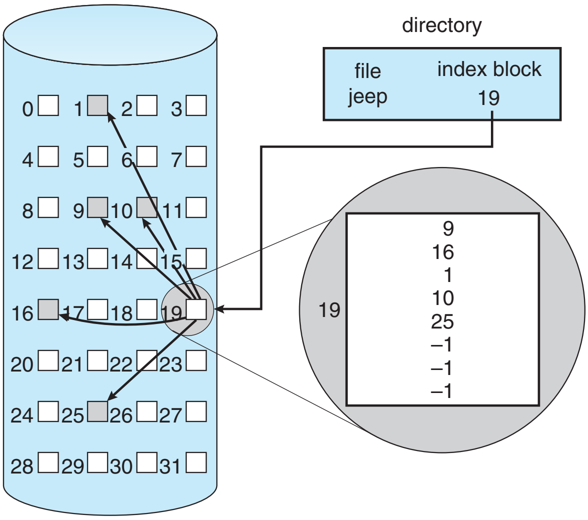


**Figure 14.7** Indexed allocation of disk space.

possible. If the index block is too small, however, it will not be able to hold enough pointers for a large file, and a mechanism will have to be available to deal with this issue. Mechanisms for this purpose include the following:

- **Linked scheme**. An index block is normally one storage block. Thus, it can be read and written directly by itself. To allow for large files, we can link together several index blocks. For example, an index block might contain a small header giving the name of the file and a set of the first 100 disk-block addresses. The next address (the last word in the index block) is `null` (for a small file) or is a pointer to another index block (for a large file).

- **Multilevel index**. A variant of linked representation uses a first-level index block to point to a set of second-level index blocks, which in turn point to the file blocks. To access a block, the operating system uses the first-level index to find a second-level index block and then uses that block to find the desired data block. This approach could be continued to a third or fourth level, depending on the desired maximum file size. With 4,096-byte blocks, we could store 1,024 four-byte pointers in an index block. Two levels of indexes allow 1,048,576 data blocks and a file size of up to 4 GB.

- **Combined scheme**. Another alternative, used in UNIX-based file systems, is to keep the first, say, 15 pointers of the index block in the file's inode. The first 12 of these pointers point to **direct blocks**; that is, they contain addresses of blocks that contain data of the file. Thus, the data for small files (of no more than 12 blocks) do not need a separate index block. If the block size is 4 KB, then up to 48 KB of data can be accessed directly. The next three pointers point to **indirect blocks**. The first points to a **single indirect block**, which is an index block containing not data but the addresses of blocks that do contain data. The second points to a **double indirect block**, which contains the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks. The last pointer contains the address of a **triple indirect block**. (A UNIX inode is shown in Figure 14.8.)

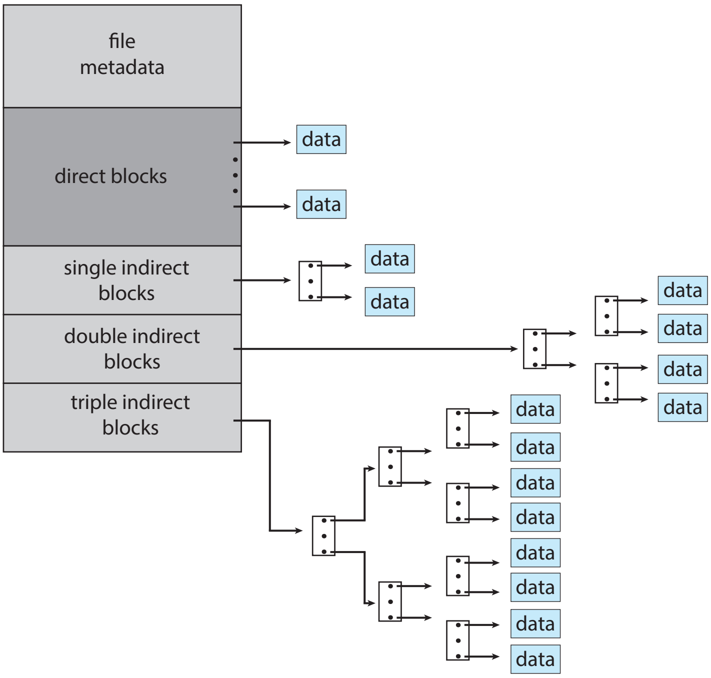


**Figure 14.8** The UNIX inode.

Under this method, the number of blocks that can be allocated to a file exceeds the amount of space addressable by the 4-byte file pointers used by many operating systems. A 32-bit file pointer reaches only $2^{32}$ bytes, or 4 GB. Many UNIX and Linux implementations now support 64-bit file pointers, which allows files and file systems to be several exbibytes in size. The ZFS file system supports 128-bit file pointers.

Indexed-allocation schemes suffer from some of the same performance problems as does linked allocation. Specifically, the index blocks can be cached in memory, but the data blocks may be spread all over a volume.

### 14.4.4 Performance

The allocation methods that we have discussed vary in their storage efficiency and data-block access times. Both are important criteria in selecting the proper method or methods for an operating system to implement.

Before selecting an allocation method, we need to determine how the systems will be used. A system with mostly sequential access should not use the same method as a system with mostly random access.

For any type of access, contiguous allocation requires only one access to get a block. Since we can easily keep the initial address of the file in memory, we can calculate immediately the address of the *i*th block (or the next block) and read it directly.

For linked allocation, we can also keep the address of the next block in memory and read it directly. This method is fine for sequential access; for direct access, however, an access to the *i*th block might require *i* block reads. This problem indicates why linked allocation should not be used for an application requiring direct access.

As a result, some systems support direct-access files by using contiguous allocation and sequential-access files by using linked allocation. For these systems, the type of access to be made must be declared when the file is created. A file created for sequential access will be linked and cannot be used for direct access. A file created for direct access will be contiguous and can support both direct access and sequential access, but its maximum length must be declared when it is created. In this case, the operating system must have appropriate data structures and algorithms to support both allocation methods. Files can be converted from one type to another by the creation of a new file of the desired type, into which the contents of the old file are copied. The old file may then be deleted and the new file renamed.

Indexed allocation is more complex. If the index block is already in memory, then the access can be made directly. However, keeping the index block in memory requires considerable space. If this memory space is not available, then we may have to read first the index block and then the desired data block. For a two-level index, two index-block reads might be necessary. For an extremely large file, accessing a block near the end of the file would require reading in all the index blocks before the needed data block finally could be read. Thus, the performance of indexed allocation depends on the index structure, on the size of the file, and on the position of the block desired.

Some systems combine contiguous allocation with indexed allocation by using contiguous allocation for small files (up to three or four blocks) and automatically switching to an indexed allocation if the file grows large. Since most files are small, and contiguous allocation is efficient for small files, average performance can be quite good.

Many other optimizations are in use. Given the disparity between CPU speed and disk speed, it is not unreasonable to add thousands of extra instructions to the operating system to save just a few disk-head movements. Furthermore, this disparity is increasing over time, to the point where hundreds of thousands of instructions could reasonably be used to optimize head movements.

For NVM devices, there are no disk head seeks, so different algorithms and optimizations are needed. Using an old algorithm that spends many CPU cycles trying to avoid a nonexistent head movement would be very inefficient. Existing file systems are being modified and new ones being created to attain maximum performance from NVM storage devices. These developments aim to reduce the instruction count and overall path between the storage device and application access to the data.

## 14.5 Free-Space Management

Since storage space is limited, we need to reuse the space from deleted files for new files, if possible. (Write-once optical disks allow only one write to any given sector, and thus reuse is not physically possible.) To keep track of free disk space, the system maintains a **free-space list**. The free-space list records all free device blocks—those not allocated to some file or directory. To create a file, we search the free-space list for the required amount of space and allocate that space to the new file. This space is then removed from the free-space list. When a file is deleted, its space is added to the free-space list. The free-space list, despite its name, is not necessarily implemented as a list, as we discuss next.

### 14.5.1 Bit Vector

Frequently, the free-space list is implemented as a **bitmap** or **bit vector**. Each block is represented by 1 bit. If the block is free, the bit is 1; if the block is allocated, the bit is 0.

For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free and the rest of the blocks are allocated. The free-space bitmap would be

001111001111110001100000011100000 ...

The main advantage of this approach is its relative simplicity and its efficiency in finding the first free block or *n* consecutive free blocks on the disk. Indeed, many computers supply bit-manipulation instructions that can be used effectively for that purpose. One technique for finding the first free block on a system that uses a bit vector to allocate space is to sequentially check each word in the bitmap to see whether that value is not 0, since a 0-valued word contains only 0 bits and represents a set of allocated blocks. The first non-0 word is scanned for the first 1 bit, which is the location of the first free block. The calculation of the block number is

(number of bits per word) × (number of 0-value words) + offset of first 1 bit.

Again, we see hardware features driving software functionality. Unfortunately, bit vectors are inefficient unless the entire vector is kept in main memory (and is written to the device containing the file system occasionally for recovery needs). Keeping it in main memory is possible for smaller devices but not necessarily for larger ones. A 1.3-GB disk with 512-byte blocks would need a bitmap of over 332 KB to track its free blocks, although clustering the blocks in groups of four reduces this number to around 83 KB per disk. A 1-TB disk with 4-KB blocks would require 32 MB ($2^{40}$ / $2^{12}$ = $2^{28}$ bits = $2^{25}$ bytes = $2^{5}$ MB) to store its bitmap. Given that disk size constantly increases, the problem with bit vectors will continue to escalate as well.

### 14.5.2 Linked List

Another approach to free-space management is to link together all the free blocks, keeping a pointer to the first free block in a special location in the file system and caching it in memory. This first block contains a pointer to the next free block, and so on. Recall our earlier example (Section 14.5.1), in which blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 were free and the rest of the blocks were allocated. In this situation, we would keep a pointer to block 2 as the first free block. Block 2 would contain a pointer to block 3, which would point to block 4, which would point to block 5, which would point to block 8, and so on (Figure 14.9). This scheme is not efficient; to traverse the list, we must read each block, which requires substantial I/O time on HDDs. Fortunately, however, traversing the free list is not a frequent action. Usually,

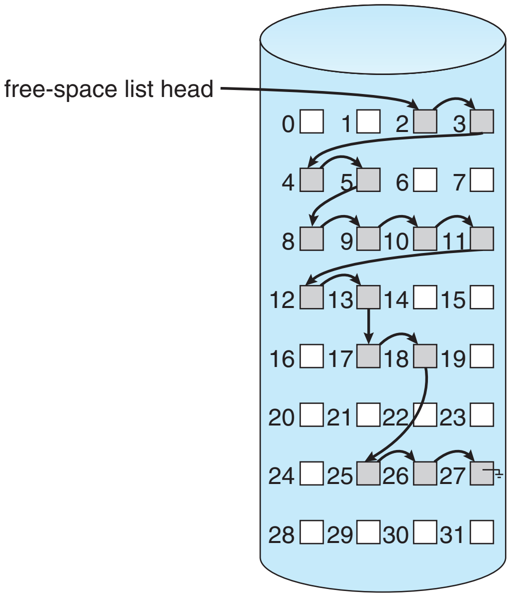


**Figure 14.9** Linked free-space list on disk.

the operating system simply needs a free block so that it can allocate that block to a file, so the first block in the free list is used. The FAT method incorporates free-block accounting into the allocation data structure. No separate method is needed.

### 14.5.3 Grouping

A modification of the free-list approach stores the addresses of $n$ free blocks in the first free block. The first $n-1$ of these blocks are actually free. The last block contains the addresses of another $n$ free blocks, and so on. The addresses of a large number of free blocks can now be found quickly, unlike the situation when the standard linked-list approach is used.

### 14.5.4 Counting

Another approach takes advantage of the fact that, generally, several contiguous blocks may be allocated or freed simultaneously, particularly when space is allocated with the contiguous-allocation algorithm or through clustering. Thus, rather than keeping a list of $n$ free block addresses, we can keep the address of the first free block and the number ($n$) of free contiguous blocks that follow the first block. Each entry in the free-space list then consists of a device address and a count. Although each entry requires more space than would a simple disk address, the overall list is shorter, as long as the count is generally greater than 1. Note that this method of tracking free space is similar to the extent method of allocating blocks. These entries can be stored in a balanced tree, rather than a linked list, for efficient lookup, insertion, and deletion.

### 14.5.5 Space Maps

Oracle's **ZFS** file system (found in Solaris and some other operating systems) was designed to encompass huge numbers of files, directories, and even file systems (in ZFS, we can create file-system hierarchies). On these scales, metadata I/O can have a large performance impact. Consider, for example, that if the free-space list is implemented as a bitmap, bitmaps must be modified both when blocks are allocated and when they are freed. Freeing 1 GB of data on a 1-TB disk could cause thousands of blocks of bitmaps to be updated, because those data blocks could be scattered over the entire disk. Clearly, the data structures for such a system could be large and inefficient.

In its management of free space, ZFS uses a combination of techniques to control the size of data structures and minimize the I/O needed to manage those structures. First, ZFS creates **metaslabs** to divide the space on the device into chunks of manageable size. A given volume may contain hundreds of metaslabs. Each metaslab has an associated space map. ZFS uses the counting algorithm to store information about free blocks. Rather than write counting structures to disk, it uses log-structured file-system techniques to record them. The space map is a log of all block activity (allocating and freeing), in time order, in counting format. When ZFS decides to allocate or free space from a metaslab, it loads the associated space map into memory in a balanced-tree structure (for very efficient operation), indexed by offset, and replays the log into that structure. The in-memory space map is then an accurate representation of the allocated and free space in the metaslab. ZFS also condenses the map as much as possible by combining contiguous free blocks into a single entry. Finally, the free-space list is updated on disk as part of the transaction-oriented operations of ZFS. During the collection and sorting phase, block requests can still occur, and ZFS satisfies these requests from the log. In essence, the log plus the balanced tree ***is*** the free list.

### 14.5.6 TRIMing Unused Blocks

HDDs and other storage media that allow blocks to be overwritten for updates need only the free list for managing free space. Blocks do not need to be treated specially when freed. A freed block typically keeps its data (but without any file pointers to the block) until the data are overwritten when the block is next allocated.

Storage devices that do not allow overwrite, such as NVM flash-based storage devices, suffer badly when these same algorithms are applied. Recall from Section 11.1.2 that such devices must be erased before they can again be written to, and that those erases must be made in large chunks (blocks, composed of pages) and take a relatively long time compared with reads or writes.

A new mechanism is needed to allow the file system to inform the storage device that a page is free and can be considered for erasure (once the block containing the page is entirely free). That mechanism varies based on the storage controller. For ATA-attached drives, it is TRIM, while for NVMe-based storage, it is the `unallocate` command. Whatever the specific controller command, this mechanism keeps storage space available for writing. Without such a capability, the storage device gets full and needs garbage collection and block erasure, leading to decreases in storage I/O write performance (known as "a write cliff").

With the TRIM mechanism and similar capabilities, the garbage collection and erase steps can occur before the device is nearly full, allowing the device to provide more consistent performance.

## 14.6 Efficiency and Performance

Now that we have discussed various block-allocation and directory-management options, we can further consider their effect on performance and efficient storage use. Disks tend to represent a major bottleneck in system performance, since they are the slowest main computer component. Even NVM devices are slow compared with CPU and main memory, so their performance must be optimized as well. In this section, we discuss a variety of techniques used to improve the efficiency and performance of secondary storage.

### 14.6.1 Efficiency

The efficient use of storage device space depends heavily on the allocation and directory algorithms in use. For instance, UNIX inodes are preallocated on a volume. Even an empty disk has a percentage of its space lost to inodes. However, by preallocating the inodes and spreading them across the volume, we improve the file system's performance. This improved performance results from the UNIX allocation and free-space algorithms, which try to keep a file's data blocks near that file's inode block to reduce seek time.

As another example, let's reconsider the clustering scheme discussed in Section 14.4, which improves file-seek and file-transfer performance at the cost of internal fragmentation. To reduce this fragmentation, BSD UNIX varies the cluster size as a file grows. Large clusters are used where they can be filled, and small clusters are used for small files and the last cluster of a file. This system is described in Appendix C.

The types of data normally kept in a file's directory (or inode) entry also require consideration. Commonly, a "last write date" is recorded to supply information to the user and to determine whether the file needs to be backed up. Some systems also keep a "last access date," so that a user can determine when the file was last read. The result of keeping this information is that, whenever the file is read, a field in the directory structure must be written to. That means the block must be read into memory, a section changed, and the block written back out to the device, because operations on secondary storage occur only in block (or cluster) chunks. So any time a file is opened for reading, its FCB must be read and written as well. This requirement can be inefficient for frequently accessed files, so we must weigh its benefit against its performance cost when designing a file system. Generally, every data item associated with a file needs to be considered for its effect on efficiency and performance.

Consider, for instance, how efficiency is affected by the size of the pointers used to access data. Most systems use either 32-bit or 64-bit pointers throughout the operating system. Using 32-bit pointers limits the size of a file to $2^{32}$, or 4 GB. Using 64-bit pointers allows very large file sizes, but 64-bit pointers require more space to store. As a result, the allocation and free-space-management methods (linked lists, indexes, and so on) use more storage space.

One of the difficulties in choosing a pointer size—or, indeed, any fixed allocation size within an operating system—is planning for the effects of changing technology. Consider that the IBM PC XT had a 10-MB hard drive and an MS-DOS FAT file system that could support only 32 MB. (Each FAT entry was 12 bits, pointing to an 8-KB cluster.) As disk capacities increased, larger disks had to be split into 32-MB partitions, because the file system could not track blocks beyond 32 MB. As hard disks with capacities of over 100 MB became common, the disk data structures and algorithms in MS-DOS had to be modified to allow larger file systems. (Each FAT entry was expanded to 16 bits and later to 32 bits.) The initial file-system decisions were made for efficiency reasons; however, with the advent of MS-DOS Version 4, millions of computer users were inconvenienced when they had to switch to the new, larger file system. Solaris's ZFS file system uses 128-bit pointers, which theoretically should never need to be extended. (The minimum mass of a device capable of storing $2^{128}$ bytes using atomic-level storage would be about 272 trillion kilograms.)

As another example, consider the evolution of the Solaris operating system. Originally, many data structures were of fixed length, allocated at system startup. These structures included the process table and the open-file table. When the process table became full, no more processes could be created. When the file table became full, no more files could be opened. The system would fail to provide services to users. Table sizes could be increased only by recompiling the kernel and rebooting the system. With later releases of Solaris, (as with modern Linux kernels) almost all kernel structures were allocated dynamically, eliminating these artificial limits on system performance. Of course, the algorithms that manipulate these tables are more complicated, and the operating system is a little slower because it must dynamically allocate and deallocate table entries; but that price is the usual one for more general functionality.

### 14.6.2 Performance

Even after the basic file-system algorithms have been selected, we can still improve performance in several ways. As was discussed in Chapter 12, storage device controllers include local memory to form an on-board cache that is large enough to store entire tracks or blocks at a time. On an HDD, once a seek is performed, the track is read into the disk cache starting at the sector under the disk head (reducing latency time). The disk controller then transfers any sector requests to the operating system. Once blocks make it from the disk controller into main memory, the operating system may cache the blocks there.

Some systems maintain a separate section of main memory for a **buffer cache**, where blocks are kept under the assumption that they will be used again shortly. Other systems cache file data using a **page cache**. The page cache uses virtual memory techniques to cache file data as pages rather than as file-system-oriented blocks. Caching file data using virtual addresses is far more efficient than caching through physical disk blocks, as accesses interface with virtual memory rather than the file system. Several systems—including Solaris, Linux, and Windows—use page caching to cache both process pages and file data. This is known as **unified virtual memory**.

Some versions of UNIX and Linux provide a **unified buffer cache**. To illustrate the benefits of the unified buffer cache, consider the two alternatives

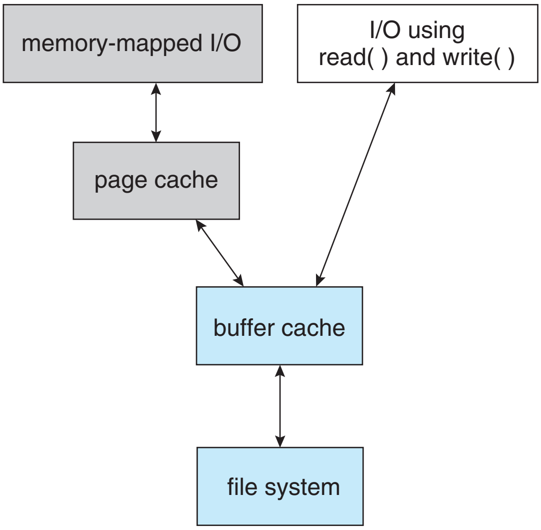


**Figure 14.10** I/O without a unified buffer cache.

for opening and accessing a file. One approach is to use memory mapping (Section 13.5); the second is to use the standard system calls `read()` and `write()`. Without a unified buffer cache, we have a situation similar to Figure 14.10. Here, the `read()` and `write()` system calls go through the buffer cache. The memory-mapping call, however, requires using two caches—the page cache and the buffer cache. A memory mapping proceeds by reading in disk blocks from the file system and storing them in the buffer cache. Because the virtual memory system does not interface with the buffer cache, the contents of the file in the buffer cache must be copied into the page cache. This situation, known as **double caching**, requires caching file-system data twice. Not only does this waste memory but it also wastes significant CPU and I/O cycles due to the extra data movement within system memory. In addition, inconsistencies between the two caches can result in corrupt files. In contrast, when a unified buffer cache is provided, both memory mapping and the `read()` and `write()` system calls use the same page cache. This has the benefit of avoiding double caching, and it allows the virtual memory system to manage file-system data. The unified buffer cache is shown in Figure 14.11.

Regardless of whether we are caching storage blocks or pages (or both), least recently used (LRU) (Section 10.4.4) seems a reasonable general-purpose algorithm for block or page replacement. However, the evolution of the Solaris page-caching algorithms reveals the difficulty in choosing an algorithm. Solaris allows processes and the page cache to share unused memory. Versions earlier than Solaris 2.5.1 made no distinction between allocating pages to a process and allocating them to the page cache. As a result, a system performing many I/O operations used most of the available memory for caching pages. Because of the high rates of I/O, the page scanner (Section 10.10.3) reclaimed pages from processes—rather than from the page cache—when free memory ran low. Solaris 2.6 and Solaris 7 optionally implemented priority paging, in which the page scanner gave priority to process pages over the page cache. Solaris 8 applied a fixed limit to process pages and the file-system page cache, prevent-

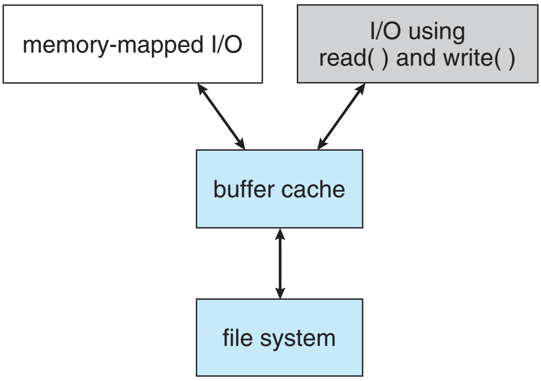


**Figure 14.11** I/O using a unified buffer cache.

ing either from forcing the other out of memory. Solaris 9 and 10 again changed the algorithms to maximize memory use and minimize thrashing.

Another issue that can affect the performance of I/O is whether writes to the file system occur synchronously or asynchronously. **Synchronous writes** occur in the order in which the storage subsystem receives them, and the writes are not buffered. Thus, the calling routine must wait for the data to reach the drive before it can proceed. In an **asynchronous write**, the data are stored in the cache, and control returns to the caller. Most writes are asynchronous. However, metadata writes, among others, can be synchronous. Operating systems frequently include a flag in the `open` system call to allow a process to request that writes be performed synchronously. For example, databases use this feature for atomic transactions, to assure that data reach stable storage in the required order.

Some systems optimize their page cache by using different replacement algorithms, depending on the access type of the file. A file being read or written sequentially should not have its pages replaced in LRU order, because the most recently used page will be used last, or perhaps never again. Instead, sequential access can be optimized by techniques known as free-behind and read-ahead. **Free-behind** removes a page from the buffer as soon as the next page is requested. The previous pages are not likely to be used again and waste buffer space. With **read-ahead**, a requested page and several subsequent pages are read and cached. These pages are likely to be requested after the current page is processed. Retrieving these data from the disk in one transfer and caching them saves a considerable amount of time. One might think that a track cache on the controller would eliminate the need for read-ahead on a multiprogrammed system. However, because of the high latency and overhead involved in making many small transfers from the track cache to main memory, performing a read-ahead remains beneficial.

The page cache, the file system, and the device drivers have some interesting interactions. When small amounts of data are written to a file, the pages are buffered in the cache, and the storage device driver sorts its output queue according to device address. These two actions allow a disk driver to minimize disk-head seeks. Unless synchronous writes are required, a process writing to disk simply writes into the cache, and the system asynchronously writes the data to disk when convenient. The user process sees very fast writes. When data are read from a disk file, the block I/O system does some read-ahead; however, writes are much more nearly asynchronous than are reads. Thus, output to the disk through the file system is often faster than is input for small transfers, counter to intuition. No matter how much buffering and caching is available, large, continuous I/O can overrun the capacity and end up bottlenecked on the device's performance. Consider writing a large movie file to a HDD. If the file is larger than the page cache (or the part of the page cache available to the process) then the page cache will fill and all I/O will occur at drive speed. Current HDDs read faster than they write, so in this instance the performance aspects are reversed from smaller I/O performance.

## 14.7 Recovery

Files and directories are kept both in main memory and on the storage volume, and care must be taken to ensure that a system failure does not result in loss of data or in data inconsistency. A system crash can cause inconsistencies among on-storage file-system data structures, such as directory structures, free-block pointers, and free FCB pointers. Many file systems apply changes to these structures in place. A typical operation, such as creating a file, can involve many structural changes within the file system on the disk. Directory structures are modified, FCBs are allocated, data blocks are allocated, and the free counts for all of these blocks are decreased. These changes can be interrupted by a crash, and inconsistencies among the structures can result. For example, the free FCB count might indicate that an FCB had been allocated, but the directory structure might not point to the FCB. Compounding this problem is the caching that operating systems do to optimize I/O performance. Some changes may go directly to storage, while others may be cached. If the cached changes do not reach the storage device before a crash occurs, more corruption is possible.

In addition to crashes, bugs in file-system implementation, device controllers, and even user applications can corrupt a file system. File systems have varying methods to deal with corruption, depending on the file-system data structures and algorithms. We deal with these issues next.

### 14.7.1 Consistency Checking

Whatever the cause of corruption, a file system must first detect the problems and then correct them. For detection, a scan of all the metadata on each file system can confirm or deny the consistency of the system. Unfortunately, this scan can take minutes or hours and should occur every time the system boots. Alternatively, a file system can record its state within the file-system metadata. At the start of any metadata change, a status bit is set to indicate that the metadata is in flux. If all updates to the metadata complete successfully, the file system can clear that bit. If, however, the status bit remains set, a consistency checker is run.

The **consistency checker**—a systems program such as `fsck` in UNIX—compares the data in the directory structure and other metadata with the state on storage and tries to fix any inconsistencies it finds. The allocation and free-space-management algorithms dictate what types of problems the checker can find and how successful it will be in fixing them. For instance, if linked allocation is used and there is a link from any block to its next block, then the entire file can be reconstructed from the data blocks, and the directory structure can be recreated. In contrast, the loss of a directory entry on an indexed allocation system can be disastrous, because the data blocks have no knowledge of one another. For this reason, some UNIX file systems cache directory entries for reads, but any write that results in space allocation, or other metadata changes, is done synchronously, before the corresponding data blocks are written. Of course, problems can still occur if a synchronous write is interrupted by a crash. Some NVM storage devices contain a battery or supercapacitor to provide enough power, even during a power loss, to write data from device buffers to the storage media so the data are not lost. But even those precautions do not protect against corruption due to a crash.

### 14.7.2 Log-Structured File Systems

Computer scientists often find that algorithms and technologies originally used in one area are equally useful in other areas. Such is the case with the database log-based recovery algorithms. These logging algorithms have been applied successfully to the problem of consistency checking. The resulting implementations are known as **log-based transaction-oriented** (or **journaling**) file systems.

Note that with the consistency-checking approach discussed in the preceding section, we essentially allow structures to break and repair them on recovery. However, there are several problems with this approach. One is that the inconsistency may be irreparable. The consistency check may not be able to recover the structures, resulting in loss of files and even entire directories. Consistency checking can require human intervention to resolve conflicts, and that is inconvenient if no human is available. The system can remain unavailable until the human tells it how to proceed. Consistency checking also takes system and clock time. To check terabytes of data, hours of clock time may be required.

The solution to this problem is to apply log-based recovery techniques to file-system metadata updates. Both NTFS and the Veritas file system use this method, and it is included in recent versions of UFS on Solaris. In fact, it is now common on many file systems including ext3, ext4, and ZFS.

Fundamentally, all metadata changes are written sequentially to a log. Each set of operations for performing a specific task is a **transaction**. Once the changes are written to this log, they are considered to be committed, and the system call can return to the user process, allowing it to continue execution. Meanwhile, these log entries are replayed across the actual file-system structures. As the changes are made, a pointer is updated to indicate which actions have completed and which are still incomplete. When an entire committed transaction is completed, and entry is made in the log indicating that. The log file is is actually a circular buffer. A **circular buffer** writes to the end of its space and then continues at the beginning, overwriting older values as it goes. We would not want the buffer to write over data that had not yet been saved, so that scenario is avoided. The log may be in a separate section of the file system or even on a separate storage device.

If the system crashes, the log file will contain zero or more transactions. Any transactions it contains were not completed to the file system, even though they were committed by the operating system, so they must now be completed. The transactions can be executed from the pointer until the work is complete so that the file-system structures remain consistent. The only problem occurs when a transaction was aborted—that is, was not committed before the system crashed. Any changes from such a transaction that were applied to the file system must be undone, again preserving the consistency of the file system. This recovery is all that is needed after a crash, eliminating any problems with consistency checking.

A side benefit of using logging on disk metadata updates is that those updates proceed much faster than when they are applied directly to the on-disk data structures. The reason is found in the performance advantage of sequential I/O over random I/O. The costly synchronous random metadata writes are turned into much less costly synchronous sequential writes to the log-structured file system's logging area. Those changes, in turn, are replayed asynchronously via random writes to the appropriate structures. The overall result is a significant gain in performance of metadata-oriented operations, such as file creation and deletion, on HDD storage.

### 14.7.3 Other Solutions

Another alternative to consistency checking is employed by Network Appliance's WAFL file system and the Solaris ZFS file system. These systems never overwrite blocks with new data. Rather, a transaction writes all data and metadata changes to new blocks. When the transaction is complete, the metadata structures that pointed to the old versions of these blocks are updated to point to the new blocks. The file system can then remove the old pointers and the old blocks and make them available for reuse. If the old pointers and blocks are kept, a **snapshot** is created; the snapshot is a view of the file system at a specific point in time (before any updates after that time were applied). This solution should require no consistency checking if the pointer update is done atomically. WAFL does have a consistency checker, however, so some failure scenarios can still cause metadata corruption. (See Section 14.8 for details of the WAFL file system.)

ZFS takes an even more innovative approach to disk consistency. Like WAFL, it never overwrites blocks. However, ZFS goes further and provides checksumming of all metadata and data blocks. This solution (when combined with RAID) assures that data are always correct. ZFS therefore has no consistency checker. (More details on ZFS are found in Section 11.8.6.)

### 14.7.4 Backup and Restore

Storage devices sometimes fail, and care must be taken to ensure that the data lost in such a failure are not lost forever. To this end, system programs can be used to **back up** data from one storage device to another, such as a magnetic tape or other secondary storage device. Recovery from the loss of an individual file, or of an entire device, may then be a matter of **restoring** the data from backup.

To minimize the copying needed, we can use information from each file's directory entry. For instance, if the backup program knows when the last backup of a file was done, and the file's last write date in the directory indicates that the file has not changed since that date, then the file does not need to be copied again. A typical backup schedule may then be as follows:

- **Day 1**. Copy to a backup medium all files from the file system. This is called a **full backup**.
- **Day 2**. Copy to another medium all files changed since day 1. This is an **incremental backup**.
- **Day 3**. Copy to another medium all files changed since day 2.

. . .

- **Day** $N$. Copy to another medium all files changed since day $N-1$. Then go back to day 1.

The new cycle can have its backup written over the previous set or onto a new set of backup media.

Using this method, we can restore an entire file system by starting restores with the full backup and continuing through each of the incremental backups. Of course, the larger the value of $N$, the greater the number of media that must be read for a complete restore. An added advantage of this backup cycle is that we can restore any file accidentally deleted during the cycle by retrieving the deleted file from the backup of the previous day.

The length of the cycle is a compromise between the amount of backup needed and the number of days covered by a restore. To decrease the number of tapes that must be read to do a restore, an option is to perform a full backup and then each day back up all files that have changed since the full backup. In this way, a restore can be done via the most recent incremental backup and the full backup, with no other incremental backups needed. The trade-off is that more files will be modified each day, so each successive incremental backup involves more files and more backup media.

A user may notice that a particular file is missing or corrupted long after the damage was done. For this reason, we usually plan to take a full backup from time to time that will be saved "forever." It is a good idea to store these permanent backups far away from the regular backups to protect against hazard, such as a fire that destroys the computer and all the backups too. In the TV show "Mr. Robot," hackers not only attacked the primary sources of banks' data but also their backup sites. Having multiple backup sites might not be a bad idea if your data are important.

## 14.8 Example: The WAFL File System

Because secondary-storage I/O has such a huge impact on system performance, file-system design and implementation command quite a lot of attention from system designers. Some file systems are general purpose, in that they can provide reasonable performance and functionality for a wide variety of file sizes, file types, and I/O loads. Others are optimized for specific tasks in an attempt to provide better performance in those areas than general-purpose file systems. The **write-anywhere file layout** (**WAFL**) from NetApp, Inc. is an example of this sort of optimization. WAFL is a powerful, elegant file system optimized for random writes.

WAFL is used exclusively on network file servers produced by NetApp and is meant for use as a distributed file system. It can provide files to clients via the NFS, CIFS, iSCSI, `ftp`, and `http` protocols, although it was designed just for NFS and CIFS. When many clients use these protocols to talk to a file server, the server may see a very large demand for random reads and an even larger demand for random writes. The NFS and CIFS protocols cache data from read operations, so writes are of the greatest concern to file-server creators.

WAFL is used on file servers that include an NVRAM cache for writes. The WAFL designers took advantage of running on a specific architecture to optimize the file system for random I/O, with a stable-storage cache in front. Ease of use is one of the guiding principles of WAFL. Its creators also designed it to include a new snapshot functionality that creates multiple read-only copies of the file system at different points in time, as we shall see.

The file system is similar to the Berkeley Fast File System, with many modifications. It is block-based and uses inodes to describe files. Each inode contains 16 pointers to blocks (or indirect blocks) belonging to the file described by the inode. Each file system has a root inode. All of the metadata lives in files. All inodes are in one file, the free-block map in another, and the free-inode map in a third, as shown in Figure 14.12. Because these are standard files, the data blocks are not limited in location and can be placed anywhere. If a file system is expanded by addition of disks, the lengths of the metadata files are automatically expanded by the file system.

Thus, a WAFL file system is a tree of blocks with the root inode as its base. To take a snapshot, WAFL creates a copy of the root inode. Any file or metadata updates after that go to new blocks rather than overwriting their existing blocks. The new root inode points to metadata and data changed as a result of these writes. Meanwhile, the snapshot (the old root inode) still points to the old blocks, which have not been updated. It therefore provides access to the file system just as it was at the instant the snapshot was made—and takes very little storage space to do so. In essence, the extra space occupied by a snapshot consists of just the blocks that have been modified since the snapshot was taken.

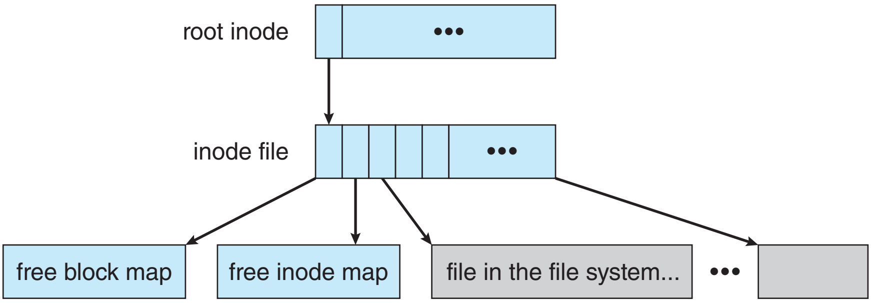
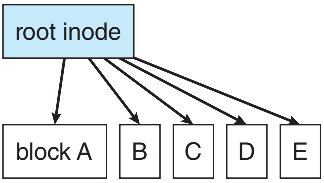


**Figure 14.12** The WAFL file layout.

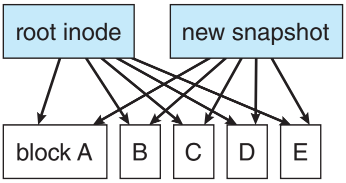
An important change from more standard file systems is that the free-block map has more than one bit per block. It is a bitmap with a bit set for each snapshot that is using the block. When all snapshots that have been using the block are deleted, the bitmap for that block is all zeros, and the block is free to be reused. Used blocks are never overwritten, so writes are very fast, because a write can occur at the free block nearest the current head location. There are many other performance optimizations in WAFL as well.

Many snapshots can exist simultaneously, so one can be taken each hour of the day and each day of the month, for example. A user with access to these snapshots can access files as they were at any of the times the snapshots were taken. The snapshot facility is also useful for backups, testing, versioning, and so on. WAFL's snapshot facility is very efficient in that it does not even require that copy-on-write copies of each data block be taken before the block is modified. Other file systems provide snapshots, but frequently with less efficiency. WAFL snapshots are depicted in Figure 14.13.

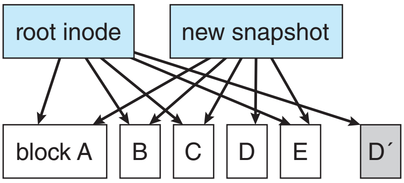
Newer versions of WAFL actually allow read–write snapshots, known as **clones**. Clones are also efficient, using the same techniques as shapshots. In this case, a read-only snapshot captures the state of the file system, and a clone refers back to that read-only snapshot. Any writes to the clone are stored in new blocks, and the clone's pointers are updated to refer to the new blocks. The original snapshot is unmodified, still giving a view into the file system as



(a) Before a snapshot.



(b) After a snapshot, before any blocks change.



(c) After block D has changed to D´.

**Figure 14.13** Snapshots in WAFL.

### *THE APPLE FILE SYSTEM*

In 2017, Apple, Inc., released a new file system to replace its 30-year-old HFS+ file system. HFS+ had been stretched to add many new features, but as usual, this process added complexity, along with lines of code, and made adding more features more difficult. Starting from scratch on a blank page allows a design to start with current technologies and methodologies and provide the exact set of features needed.

**Apple File System** (**APFS**) is a good example of such a design. Its goal is to run on all current Apple devices, from the Apple Watch through the iPhone to the Mac computers. Creating a file system that works in watchOS, I/Os, tvOS, and macOS is certainly a challenge. APFS is feature-rich, including 64-bit pointers, clones for files and directories, snapshots, space sharing, fast directory sizing, atomic safe-save primitives, copy-on-write design, encryption (single- and multi-key), and I/O coalescing. It understands NVM as well as HDD storage.

Most of these features we've discussed, but there are a few new concepts worth exploring. **Space sharing** is a ZFS-like feature in which storage is available as one or more large free spaces (**containers**) from which file systems can draw allocations (allowing APFS-formatted volumes to grow and shrink). **Fast directory sizing** provides quick used-space calculation and updating. **Atomic safe-save** is a primitive (available via API, not via file-system commands) that performs renames of files, bundles of files, and directories as single atomic operations. I/O coalescing is an optimization for NVM devices in which several small writes are gathered together into a large write to optimize write performance.

Apple chose not to implement RAID as part of the new APFS, instead depending on the existing Apple RAID volume mechanism for software RAID. APFS is also compatible with HFS+, allowing easy conversion for existing deployments.

it was before the clone was updated. Clones can also be promoted to replace the original file system; this involves throwing out all of the old pointers and any associated old blocks. Clones are useful for testing and upgrades, as the original version is left untouched and the clone deleted when the test is done or if the upgrade fails.

Another feature that naturally results from the WAFL file system implementation is **replication,** the duplication and synchronization of a set of data over a network to another system. First, a snapshot of a WAFL file system is duplicated to another system. When another snapshot is taken on the source system, it is relatively easy to update the remote system just by sending over all blocks contained in the new snapshot. These blocks are the ones that have changed between the times the two snapshots were taken. The remote system adds these blocks to the file system and updates its pointers, and the new system then is a duplicate of the source system as of the time of the second snapshot. Repeating this process maintains the remote system as a nearly up-to-date copy of the first system. Such replication is used for disaster recovery. Should the first system be destroyed, most of its data are available for use on the remote system.

Finally, note that the ZFS file system supports similarly efficient snapshots, clones, and replication, and those features are becoming more common in various file systems as time goes by.

## 14.9 Summary

- Most file systems reside on secondary storage, which is designed to hold a large amount of data permanently. The most common secondary-storage medium is the disk, but the use of NVM devices is increasing.
- Storage devices are segmented into partitions to control media use and to allow multiple, possibly varying, file systems on a single device. These file systems are mounted onto a logical file system architecture to make them available for use.
- File systems are often implemented in a layered or modular structure. The lower levels deal with the physical properties of storage devices and communicating with them. Upper levels deal with symbolic file names and logical properties of files.
- The various files within a file system can be allocated space on the storage device in three ways: through contiguous, linked, or indexed allocation. Contiguous allocation can suffer from external fragmentation. Direct access is very inefficient with linked allocation. Indexed allocation may require substantial overhead for its index block. These algorithms can be optimized in many ways. Contiguous space can be enlarged through extents to increase flexibility and to decrease external fragmentation. Indexed allocation can be done in clusters of multiple blocks to increase throughput and to reduce the number of index entries needed. Indexing in large clusters is similar to contiguous allocation with extents.
- Free-space allocation methods also influence the efficiency of disk-space use, the performance of the file system, and the reliability of secondary storage. The methods used include bit vectors and linked lists. Optimizations include grouping, counting, and the FAT, which places the linked list in one contiguous area.
- Directory-management routines must consider efficiency, performance, and reliability. A hash table is a commonly used method, as it is fast and efficient. Unfortunately, damage to the table or a system crash can result in inconsistency between the directory information and the disk's contents.
- A consistency checker can be used to repair damaged file-system structures. Operating-system backup tools allow data to be copied to magnetic tape or other storage devices, enabling the user to recover from data loss or even entire device loss due to hardware failure, operating system bug, or user error.
- Due to the fundamental role that file systems play in system operation, their performance and reliability are crucial. Techniques such as log structures and caching help improve performance, while log structures and RAID improve reliability. The WAFL file system is an example of optimization of performance to match a specific I/O load.

## Practice Exercises

**14.1** Consider a file currently consisting of 100 blocks. Assume that the file-control block (and the index block, in the case of indexed allocation) is already in memory. Calculate how many disk I/O operations are required for contiguous, linked, and indexed (single-level) allocation strategies, if, for one block, the following conditions hold. In the contiguous-allocation case, assume that there is no room to grow at the beginning but there is room to grow at the end. Also assume that the block information to be added is stored in memory.

a. The block is added at the beginning.

b. The block is added in the middle.

c. The block is added at the end.

d. The block is removed from the beginning.

e. The block is removed from the middle.

f. The block is removed from the end.

**14.2** Why must the bit map for file allocation be kept on mass storage, rather than in main memory?

**14.3** Consider a system that supports the strategies of contiguous, linked, and indexed allocation. What criteria should be used in deciding which strategy is best utilized for a particular file?

**14.4** One problem with contiguous allocation is that the user must preallocate enough space for each file. If the file grows to be larger than the space allocated for it, special actions must be taken. One solution to this problem is to define a file structure consisting of an initial contiguous area of a specified size. If this area is filled, the operating system automatically defines an overflow area that is linked to the initial contiguous area. If the overflow area is filled, another overflow area is allocated. Compare this implementation of a file with the standard contiguous and linked implementations.

**14.5** How do caches help improve performance? Why do systems not use more or larger caches if they are so useful?

**14.6** Why is it advantageous to the user for an operating system to dynamically allocate its internal tables? What are the penalties to the operating system for doing so?

## Further Reading

The internals of the BSD UNIX system are covered in full in [McKusick et al. (2015)]. Details concerning file systems for Linux can be found in [Love (2010)]. The Google file system is described in [Ghemawat et al. (2003)]. FUSE can be found at http://fuse.sourceforge.net.

Log-structured file organizations for enhancing both performance and consistency are discussed in [Rosenblum and Ousterhout (1991)], [Seltzer et al. (1993)], and [Seltzer et al. (1995)]. Log-structured designs for networked file systems are proposed in [Hartman and Ousterhout (1995)] and [Thekkath et al. (1997)].

The ZFS source code for space maps can be found at http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/uts/common/ fs/zfs/space_map.c.

ZFS documentation can be found at http://www.opensolaris.org/os/community/ZFS/docs.

The NTFS file system is explained in [Solomon (1998)], the Ext3 file system used in Linux is described in [Mauerer (2008)], and the WAFL file system is covered in [Hitz et al. (1995)].

## Bibliography

**[Ghemawat et al. (2003)]** S. Ghemawat, H. Gobioff, and S.-T. Leung, “The Google File System”, *Proceedings of the ACM Symposium on Operating Systems Principles* (2003).

**[Hartman and Ousterhout (1995)]** J. H. Hartman and J. K. Ousterhout, “The Zebra Striped Network File System”, *ACM Transactions on Computer Systems*, Volume 13, Number 3 (1995), pages 274–310.

**[Hitz et al. (1995)]** D. Hitz, J. Lau, and M. Malcolm, “File System Design for an NFS File Server Appliance”, Technical report, NetApp (1995).

**[Love (2010)]** R. Love, *Linux Kernel Development*, Third Edition, Developer’s Library (2010).

**[Mauerer (2008)]** W. Mauerer, *Professional Linux Kernel Architecture*, John Wiley and Sons (2008).

**[McKusick et al. (2015)]** M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson, *The Design and Implementation of the FreeBSD UNIX Operating System–Second Edition*, Pearson (2015).

**[Rosenblum and Ousterhout (1991)]** M. Rosenblum and J. K. Ousterhout, “The Design and Implementation of a Log-Structured File System”, *Proceedings of the ACM Symposium on Operating Systems Principles* (1991), pages 1–15.

**[Seltzer et al. (1993)]** M. I. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin, “An Implementation of a Log-Structured File System for UNIX”, *USENIX Winter* (1993), pages 307–326.

**[Seltzer et al. (1995)]** M. I. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. N. Padmanabhan, “File System Logging Versus Clustering: A Performance Comparison”, *USENIX Winter* (1995), pages 249–264.

**[Solomon (1998)]** D. A. Solomon, *Inside Windows NT*, Second Edition, Microsoft Press (1998).

**[Thekkath et al. (1997)]** C. A. Thekkath, T. Mann, and E. K. Lee, “Frangipani: A Scalable Distributed File System”, *Symposium on Operating Systems Principles* (1997), pages 224–237.

# Chapter 14 Exercises

**14.7** Consider a file system that uses a modified contiguous-allocation scheme with support for extents. A file is a collection of extents, with each extent corresponding to a contiguous set of blocks. A key issue in such systems is the degree of variability in the size of the extents. What are the advantages and disadvantages of the following schemes?

a. All extents are of the same size, and the size is predetermined.

b. Extents can be of any size and are allocated dynamically.

c. Extents can be of a few fixed sizes, and these sizes are predetermined.

**14.8** Contrast the performance of the three techniques for allocating disk blocks (contiguous, linked, and indexed) for both sequential and random file access.

**14.9** What are the advantages of the variant of linked allocation that uses a FAT to chain together the blocks of a file?

**14.10** Consider a system where free space is kept in a free-space list.

a. Suppose that the pointer to the free-space list is lost. Can the system reconstruct the free-space list? Explain your answer.

b. Consider a file system similar to the one used by UNIX with indexed allocation. How many disk I/O operations might be required to read the contents of a small local file at `/a/b/c`? Assume that none of the disk blocks is currently being cached.

c. Suggest a scheme to ensure that the pointer is never lost as a result of memory failure.

**14.11** Some file systems allow disk storage to be allocated at different levels of granularity. For instance, a file system could allocate 4 KB of disk space as a single 4-KB block or as eight 512-byte blocks. How could we take advantage of this flexibility to improve performance? What modifications would have to be made to the free-space management scheme in order to support this feature?

**14.12** Discuss how performance optimizations for file systems might result in difficulties in maintaining the consistency of the systems in the event of computer crashes.

**14.13** Discuss the advantages and disadvantages of supporting links to files that cross mount points (that is, the file link refers to a file that is stored in a different volume).

**14.14** Consider a file system on a disk that has both logical and physical block sizes of 512 bytes. Assume that the information about each file is already in memory. For each of the three allocation strategies (contiguous, linked, and indexed), answer these questions:

a. How is the logical-to-physical address mapping accomplished in this system? (For the indexed allocation, assume that a file is always less than 512 blocks long.)

b. If we are currently at logical block 10 (the last block accessed was block 10) and want to access logical block 4, how many physical blocks must be read from the disk?

**14.15** Consider a file system that uses inodes to represent files. Disk blocks are 8 KB in size, and a pointer to a disk block requires 4 bytes. This file system has 12 direct disk blocks, as well as single, double, and triple indirect disk blocks. What is the maximum size of a file that can be stored in this file system?

**14.16** Fragmentation on a storage device can be eliminated through compaction. Typical disk devices do not have relocation or base registers (such as those used when memory is to be compacted), so how can we relocate files? Give three reasons why compacting and relocating files are often avoided.

**14.17** Explain why logging metadata updates ensures recovery of a file system after a file-system crash.

**14.18** Consider the following backup scheme:

- **Day 1**. Copy to a backup medium all files from the disk.
- **Day 2**. Copy to another medium all files changed since day 1.
- **Day 3**. Copy to another medium all files changed since day 1.

This differs from the schedule given in Section 14.7.4 by having all subsequent backups copy all files modified since the first full backup. What are the benefits of this system over the one in Section 14.7.4? What are the drawbacks? Are restore operations made easier or more difficult? Explain your answer.

**14.19** Discuss the advantages and disadvantages of associating with remote file systems (stored on file servers) a set of failure semantics different from those associated with local file systems.

**14.20** What are the implications of supporting UNIX consistency semantics for shared access to files stored on remote file systems?

CHAPTER 15

# File-System Internals

As we saw in Chapter 13, the file system provides the mechanism for on-line storage and access to file contents, including data and programs. This chapter is primarily concerned with the internal structures and operations of file systems. We explore in detail ways to structure file use, to allocate storage space, to recover freed space, to track the locations of data, and to interface other parts of the operating system to secondary storage.

### CHAPTER OBJECTIVES

- Delve into the details of file systems and their implementation.
- Explore booting and file sharing.
- Describe remote file systems, using NFS as an example.

## 15.1 File Systems

Certainly, no general-purpose computer stores just one file. There are typically thousands, millions, even billions of files within a computer. Files are stored on random-access storage devices, including hard disk drives, optical disks, and nonvolatile memory devices.

As you have seen in the preceding chapters, a general-purpose computer system can have multiple storage devices, and those devices can be sliced up into partitions, which hold volumes, which in turn hold file systems. Depending on the volume manager, a volume may span multiple partitions as well. Figure 15.1 shows a typical file-system organization.

Computer systems may also have varying numbers of file systems, and the file systems may be of varying types. For example, a typical Solaris system may have dozens of file systems of a dozen different types, as shown in the file-system list in Figure 15.2.

In this book, we consider only general-purpose file systems. It is worth noting, though, that there are many special-purpose file systems. Consider the types of file systems in the Solaris example mentioned above:

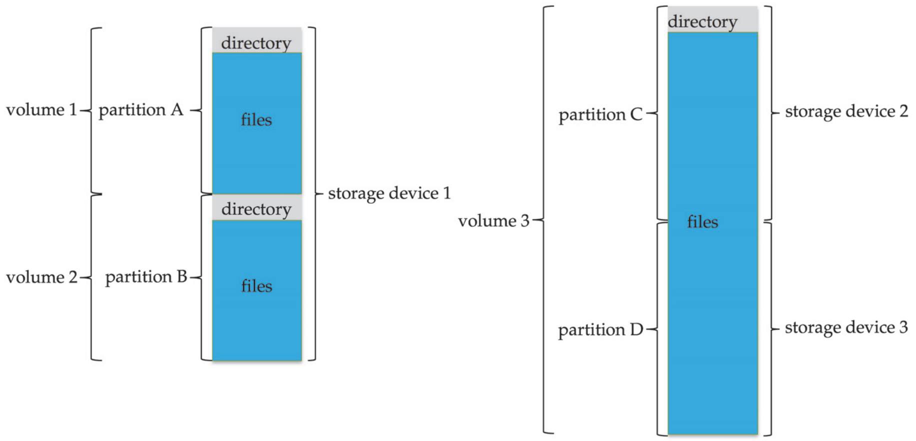


**Figure 15.1** A typical storage device organization.

- **tmpfs**—a "temporary" file system that is created in volatile main memory and has its contents erased if the system reboots or crashes
- **objfs**—a "virtual" file system (essentially an interface to the kernel that looks like a file system) that gives debuggers access to kernel symbols
- **ctfs**—a virtual file system that maintains "contract" information to manage which processes start when the system boots and must continue to run during operation
- **lofs**—a "loop back" file system that allows one file system to be accessed in place of another one
- **procfs**—a virtual file system that presents information on all processes as a file system
- **ufs, zfs**—general-purpose file systems

The file systems of computers, then, can be extensive. Even within a file system, it is useful to segregate files into groups and manage and act on those groups. This organization involves the use of directories (see Section 14.3).

## 15.2 File-System Mounting

Just as a file must be opened before it can be used, a file system must be mounted before it can be available to processes on the system. More specifically, the directory structure may be built out of multiple file-system-containing volumes, which must be mounted to make them available within the file-system name space.

The mount procedure is straightforward. The operating system is given the name of the device and the **mount point**—the location within the file structure where the file system is to be attached. Some operating systems require that a file-system type be provided, while others inspect the structures of the device

| | |
|---|---|
| / | ufs |
| /devices | devfs |
| /dev | dev |
| /system/contract | ctfs |
| /proc | proc |
| /etc/mnttab | mntfs |
| /etc/svc/volatile | tmpfs |
| /system/object | objfs |
| /lib/libc.so.1 | lofs |
| /dev/fd | fd |
| /var | ufs |
| /tmp | tmpfs |
| /var/run | tmpfs |
| /opt | ufs |
| /zpbge | zfs |
| /zpbge/backup | zfs |
| /export/home | zfs |
| /var/mail | zfs |
| /var/spool/mqueue | zfs |
| /zpbg | zfs |
| /zpbg/zones | zfs |

**Figure 15.2** Solaris file systems.

and determine the type of file system. Typically, a mount point is an empty directory. For instance, on a UNIX system, a file system containing a user's home directories might be mounted as `/home`; then, to access the directory structure within that file system, we could precede the directory names with `/home`, as in `/home/jane`. Mounting that file system under `/users` would result in the path name `/users/jane`, which we could use to reach the same directory.

Next, the operating system verifies that the device contains a valid file system. It does so by asking the device driver to read the device directory and verifying that the directory has the expected format. Finally, the operating system notes in its directory structure that a file system is mounted at the specified mount point. This scheme enables the operating system to traverse its directory structure, switching among file systems, and even file systems of varying types, as appropriate.

To illustrate file mounting, consider the file system depicted in Figure 15.3, where the triangles represent subtrees of directories that are of interest. Figure 15.3(a) shows an existing file system, while Figure 15.3(b) shows an unmounted volume residing on `/device/dsk`. At this point, only the files on the existing file system can be accessed. Figure 15.4 shows the effects of mounting the volume residing on `/device/dsk` over `/users`. If the volume is unmounted, the file system is restored to the situation depicted in Figure 15.3.

Systems impose semantics to clarify functionality. For example, a system may disallow a mount over a directory that contains files; or it may make the

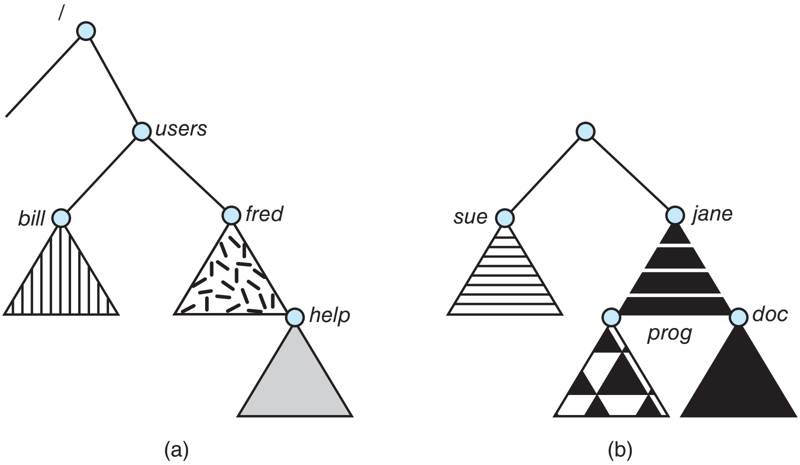


**Figure 15.3** File system. (a) Existing system. (b) Unmounted volume.

mounted file system available at that directory and obscure the directory's existing files until the file system is unmounted, terminating the use of the file system and allowing access to the original files in that directory. As another example, a system may allow the same file system to be mounted repeatedly, at different mount points; or it may only allow one mount per file system.

Consider the actions of the macOS operating system. Whenever the system encounters a disk for the first time (either at boot time or while the system is running), the macOS operating system searches for a file system on the device. If it finds one, it automatically mounts the file system under the `/Volumes` directory, adding a folder icon labeled with the name of the file system (as stored in the device directory). The user is then able to click on the icon and thus display the newly mounted file system.

The Microsoft Windows family of operating systems maintains an extended two-level directory structure, with devices and volumes assigned drive letters. Each volume has a general graph directory structure associated

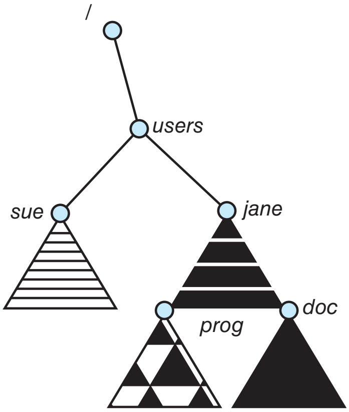


**Figure 15.4** Volume mounted at /users.

with its drive letter. The path to a specific file takes the form `drive-letter:\path\to\file`. The more recent versions of Windows allow a file system to be mounted anywhere in the directory tree, just as UNIX does. Windows operating systems automatically discover all devices and mount all located file systems at boot time. In some systems, like UNIX, the mount commands are explicit. A system configuration file contains a list of devices and mount points for automatic mounting at boot time, but other mounts may be executed manually.

Issues concerning file system mounting are further discussed in Section 15.3 and in Section C.7.5.

## 15.3 Partitions and Mounting

The layout of a disk can have many variations, depending on the operating system and volume management software. A disk can be sliced into multiple partitions, or a volume can span multiple partitions on multiple disks. The former layout is discussed here, while the latter, which is more appropriately considered a form of RAID, is covered in Section 11.8.

Each partition can be either "raw," containing no file system, or "cooked," containing a file system. **Raw disk** is used where no file system is appropriate. UNIX swap space can use a raw partition, for example, since it uses its own format on disk and does not use a file system. Likewise, some databases use raw disk and format the data to suit their needs. Raw disk can also hold information needed by disk RAID systems, such as bitmaps indicating which blocks are mirrored and which have changed and need to be mirrored. Similarly, raw disk can contain a miniature database holding RAID configuration information, such as which disks are members of each RAID set. Raw disk use is discussed in Section 11.5.1.

If a partition contains a file system that is bootable—that has a properly installed and configured operating system—then the partition also needs boot information, as described in Section 11.5.2. This information has its own format, because at boot time the system does not have the file-system code loaded and therefore cannot interpret the file-system format. Rather, boot information is usually a sequential series of blocks loaded as an image into memory. Execution of the image starts at a predefined location, such as the first byte. This image, the **bootstrap loader**, in turn knows enough about the file-system structure to be able to find and load the kernel and start it executing.

The boot loader can contain more than the instructions for booting a specific operating system. For instance, many systems can be **dual-booted**, allowing us to install multiple operating systems on a single system. How does the system know which one to boot? A boot loader that understands multiple file systems and multiple operating systems can occupy the boot space. Once loaded, it can boot one of the operating systems available on the drive. The drive can have multiple partitions, each containing a different type of file system and a different operating system. Note that if the boot loader does not understand a particular file-system format, an operating system stored on that file system is not bootable. This is one of the reasons only some file systems are supported as root file systems for any given operating system.

The **root partition** selected by the boot loader, which contains the operating-system kernel and sometimes other system files, is mounted at boot time. Other volumes can be automatically mounted at boot or manually mounted later, depending on the operating system. As part of a successful mount operation, the operating system verifies that the device contains a valid file system. It does so by asking the device driver to read the device directory and verifying that the directory has the expected format. If the format is invalid, the partition must have its consistency checked and possibly corrected, either with or without user intervention. Finally, the operating system notes in its in-memory mount table that a file system is mounted, along with the type of the file system. The details of this function depend on the operating system.

Microsoft Windows–based systems mount each volume in a separate name space, denoted by a letter and a colon, as mentioned earlier. To record that a file system is mounted at `F:`, for example, the operating system places a pointer to the file system in a field of the device structure corresponding to `F:`. When a process specifies the driver letter, the operating system finds the appropriate file-system pointer and traverses the directory structures on that device to find the specified file or directory. Later versions of Windows can mount a file system at any point within the existing directory structure.

On UNIX, file systems can be mounted at any directory. Mounting is implemented by setting a flag in the in-memory copy of the inode for that directory. The flag indicates that the directory is a mount point. A field then points to an entry in the mount table, indicating which device is mounted there. The mount table entry contains a pointer to the superblock of the file system on that device. This scheme enables the operating system to traverse its directory structure, switching seamlessly among file systems of varying types.

## 15.4 File Sharing

The ability to share files is very desirable for users who want to collaborate and to reduce the effort required to achieve a computing goal. Therefore, user-oriented operating systems must accommodate the need to share files in spite of the inherent difficulties.

In this section, we examine more aspects of file sharing. We begin by discussing general issues that arise when multiple users share files. Once multiple users are allowed to share files, the challenge is to extend sharing to multiple file systems, including remote file systems; we discuss that challenge as well. Finally, we consider what to do about conflicting actions occurring on shared files. For instance, if multiple users are writing to a file, should all the writes be allowed to occur, or should the operating system protect the users' actions from one another?

### 15.4.1 Multiple Users

When an operating system accommodates multiple users, the issues of file sharing, file naming, and file protection become preeminent. Given a directory structure that allows files to be shared by users, the system must mediate the file sharing. The system can either allow a user to access the files of other users by default or require that a user specifically grant access to the files. These are the issues of access control and protection, which are covered in Section 13.4.

To implement sharing and protection, the system must maintain more file and directory attributes than are needed on a single-user system. Although many approaches have been taken to meet this requirement, most systems have evolved to use the concepts of file (or directory) **owner** (or **user**) and **group**. The owner is the user who can change attributes and grant access and who has the most control over the file. The group attribute defines a subset of users who can share access to the file. For example, the owner of a file on a UNIX system can issue all operations on a file, while members of the file's group can execute one subset of those operations, and all other users can execute another subset of operations. Exactly which operations can be executed by group members and other users is definable by the file's owner.

The owner and group IDs of a given file (or directory) are stored with the other file attributes. When a user requests an operation on a file, the user ID can be compared with the owner attribute to determine if the requesting user is the owner of the file. Likewise, the group IDs can be compared. The result indicates which permissions are applicable. The system then applies those permissions to the requested operation and allows or denies it.

Many systems have multiple local file systems, including volumes of a single disk or multiple volumes on multiple attached disks. In these cases, the ID checking and permission matching are straightforward, once the file systems are mounted. But consider an external disk that can be moved between systems. What if the IDs on the systems are different? Care must be taken to be sure that IDs match between systems when devices move between them or that file ownership is reset when such a move occurs. (For example, we can create a new user ID and set all files on the portable disk to that ID, to be sure no files are accidentally accessible to existing users.)

## 15.5 Virtual File Systems

As we've seen, modern operating systems must concurrently support multiple types of file systems. But how does an operating system allow multiple types of file systems to be integrated into a directory structure? And how can users seamlessly move between file-system types as they navigate the file-system space? We now discuss some of these implementation details.

An obvious but suboptimal method of implementing multiple types of file systems is to write directory and file routines for each type. Instead, however, most operating systems, including UNIX, use object-oriented techniques to simplify, organize, and modularize the implementation. The use of these methods allows very dissimilar file-system types to be implemented within the same structure, including network file systems, such as NFS. Users can access files contained within multiple file systems on the local drive or even on file systems available across the network.

Data structures and procedures are used to isolate the basic system-call functionality from the implementation details. Thus, the file-system implementation consists of three major layers, as depicted schematically in Figure 15.5. The first layer is the file-system interface, based on the `open()`, `read()`, `write()`, and `close()` calls and on file descriptors.

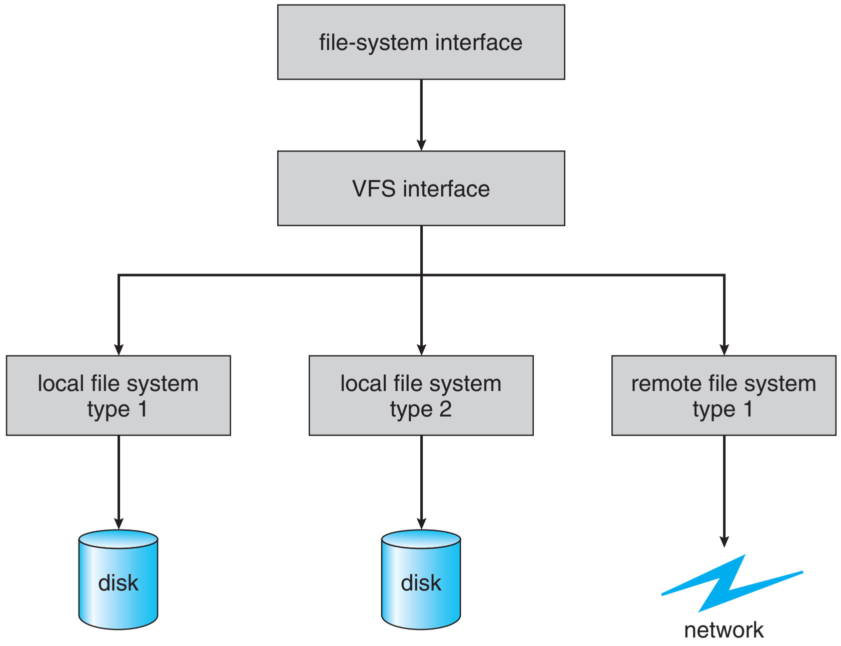


**Figure 15.5** Schematic view of a virtual file system.

The second layer is called the **virtual file system** (**VFS**) layer. The VFS layer serves two important functions:

1. It separates file-system-generic operations from their implementation by defining a clean VFS interface. Several implementations for the VFS interface may coexist on the same machine, allowing transparent access to different types of file systems mounted locally.
2. It provides a mechanism for uniquely representing a file throughout a network. The VFS is based on a file-representation structure, called a **vnode**, that contains a numerical designator for a network-wide unique file. (UNIX inodes are unique within only a single file system.) This network-wide uniqueness is required for support of network file systems. The kernel maintains one vnode structure for each active node (file or directory).

Thus, the VFS distinguishes local files from remote ones, and local files are further distinguished according to their file-system types.

The VFS activates file-system-specific operations to handle local requests according to their file-system types and calls the NFS protocol procedures (or other protocol procedures for other network file systems) for remote requests. File handles are constructed from the relevant vnodes and are passed as arguments to these procedures. The layer implementing the file-system type or the remote-file-system protocol is the third layer of the architecture.

Let's briefly examine the VFS architecture in Linux. The four main object types defined by the Linux VFS are:

- The **inode object**, which represents an individual file
- The **file object**, which represents an open file
- The **superblock object**, which represents an entire file system
- The **dentry object**, which represents an individual directory entry

For each of these four object types, the VFS defines a set of operations that may be implemented. Every object of one of these types contains a pointer to a function table. The function table lists the addresses of the actual functions that implement the defined operations for that particular object. For example, an abbreviated API for some of the operations for the file object includes:

- `int open(. . .)`—Open a file.
- `int close(. . .)`—Close an already-open file.
- `ssize_t read(. . .)`—Read from a file.
- `ssize_t write(. . .)`—Write to a file.
- `int mmap(. . .)`—Memory-map a file.

An implementation of the file object for a specific file type is required to implement each function specified in the definition of the file object. (The complete definition of the file object is specified in the file `struct file_operations`, which is located in the file `/usr/include/linux/fs.h`.)

Thus, the VFS software layer can perform an operation on one of these objects by calling the appropriate function from the object's function table, without having to know in advance exactly what kind of object it is dealing with. The VFS does not know, or care, whether an inode represents a disk file, a directory file, or a remote file. The appropriate function for that file's `read()` operation will always be at the same place in its function table, and the VFS software layer will call that function without caring how the data are actually read.

## 15.6 Remote File Systems

With the advent of networks (Chapter 19), communication among remote computers became possible. Networking allows the sharing of resources spread across a campus or even around the world. One obvious resource to share is data in the form of files.

Through the evolution of network and file technology, remote file-sharing methods have changed. The first implemented method involves manually transferring files between machines via programs like `ftp`. The second major method uses a **distributed file system** (**DFS**), in which remote directories are visible from a local machine. In some ways, the third method, the **World Wide Web**, is a reversion to the first. A browser is needed to gain access to the remote files, and separate operations (essentially a wrapper for `ftp`) are used to transfer files. Increasingly, cloud computing (Section 1.10.5) is being used for file sharing as well.

`ftp` is used for both anonymous and authenticated access. **Anonymous access** allows a user to transfer files without having an account on the remote system. The World Wide Web uses anonymous file exchange almost exclusively. DFS involves a much tighter integration between the machine that is accessing the remote files and the machine providing the files. This integration adds complexity, as we describe in this section.

### 15.6.1 The Client–Server Model

Remote file systems allow a computer to mount one or more file systems from one or more remote machines. In this case, the machine containing the files is the **server**, and the machine seeking access to the files is the **client**. The client–server relationship is common with networked machines. Generally, the server declares that a resource is available to clients and specifies exactly which resource (in this case, which files) and exactly which clients. A server can serve multiple clients, and a client can use multiple servers, depending on the implementation details of a given client–server facility.

The server usually specifies the available files on a volume or directory level. Client identification is more difficult. A client can be specified by a network name or other identifier, such as an IP address, but these can be **spoofed**, or imitated. As a result of spoofing, an unauthorized client could be allowed access to the server. More secure solutions include secure authentication of the client via encrypted keys. Unfortunately, with security come many challenges, including ensuring compatibility of the client and server (they must use the same encryption algorithms) and security of key exchanges (intercepted keys could again allow unauthorized access). Because of the difficulty of solving these problems, unsecure authentication methods are most commonly used.

In the case of UNIX and its network file system (NFS), authentication takes place via the client networking information, by default. In this scheme, the user's IDs on the client and server must match. If they do not, the server will be unable to determine access rights to files. Consider the example of a user who has an ID of 1000 on the client and 2000 on the server. A request from the client to the server for a specific file will not be handled appropriately, as the server will determine if user 1000 has access to the file rather than basing the determination on the real user ID of 2000. Access is thus granted or denied based on incorrect authentication information. The server must trust the client to present the correct user ID. Note that the NFS protocols allow many-to-many relationships. That is, many servers can provide files to many clients. In fact, a given machine can be both a server to some NFS clients and a client of other NFS servers.

Once the remote file system is mounted, file operation requests are sent on behalf of the user across the network to the server via the DFS protocol. Typically, a file-open request is sent along with the ID of the requesting user. The server then applies the standard access checks to determine if the user has credentials to access the file in the mode requested. The request is either allowed or denied. If it is allowed, a file handle is returned to the client application, and the application then can perform read, write, and other operations on the file. The client closes the file when access is completed. The operating system may apply semantics similar to those for a local file-system mount or may use different semantics.

### 15.6.2 Distributed Information Systems

To make client–server systems easier to manage, **distributed information systems**, also known as **distributed naming services**, provide unified access to the information needed for remote computing. The **domain name system** (**DNS**) provides host-name-to-network-address translations for the entire Internet. Before DNS became widespread, files containing the same information were sent via e-mail or `ftp` between all networked hosts. Obviously, this methodology was not scalable! DNS is further discussed in Section 19.3.1.

Other distributed information systems provide ***user name/password/user ID/group ID*** space for a distributed facility. UNIX systems have employed a wide variety of distributed information methods. Sun Microsystems (now part of Oracle Corporation) introduced **yellow pages** (since renamed **network information service**, or **NIS**), and most of the industry adopted its use. It centralizes storage of user names, host names, printer information, and the like. Unfortunately, it uses unsecure authentication methods, including sending user passwords unencrypted (in clear text) and identifying hosts by IP address. Sun's NIS+ was a much more secure replacement for NIS but was much more complicated and was not widely adopted.

In the case of Microsoft's **common Internet file system** (**CIFS**), network information is used in conjunction with user authentication (user name and password) to create a network login that the server uses to decide whether to allow or deny access to a requested file system. For this authentication to be valid, the user names must match from machine to machine (as with NFS). Microsoft uses **active directory** as a distributed naming structure to provide a single name space for users. Once established, the distributed naming facility is used by all clients and servers to authenticate users via Microsoft's version of the **Kerberos** network authentication protocol (https://web.mit.edu/kerberos/).

The industry is moving toward use of the **lightweight directory-access protocol** (**LDAP**) as a secure distributed naming mechanism. In fact, active directory is based on LDAP. Oracle Solaris and most other major operating systems include LDAP and allow it to be employed for user authentication as well as system-wide retrieval of information, such as availability of printers. Conceivably, one distributed LDAP directory could be used by an organization to store all user and resource information for all the organization's computers. The result would be secure single sign-on for users, who would enter their authentication information once for access to all computers within the organization. It would also ease system-administration efforts by combining, in one location, information that is currently scattered in various files on each system or in different distributed information services.

### 15.6.3 Failure Modes

Local file systems can fail for a variety of reasons, including failure of the drive containing the file system, corruption of the directory structure or other disk-management information (collectively called **metadata**), disk-controller failure, cable failure, and host-adapter failure. User or system-administrator failure can also cause files to be lost or entire directories or volumes to be deleted. Many of these failures will cause a host to crash and an error condition to be displayed, and human intervention may be required to repair the damage.

Remote file systems have even more failure modes. Because of the complexity of network systems and the required interactions between remote machines, many more problems can interfere with the proper operation of remote file systems. In the case of networks, the network can be interrupted between two hosts. Such interruptions can result from hardware failure, poor hardware configuration, or networking implementation issues. Although some networks have built-in resiliency, including multiple paths between hosts, many do not. Any single failure can thus interrupt the flow of DFS commands.

Consider a client in the midst of using a remote file system. It has files open from the remote host; among other activities, it may be performing directory lookups to open files, reading or writing data to files, and closing files. Now consider a partitioning of the network, a crash of the server, or even a scheduled shutdown of the server. Suddenly, the remote file system is no longer reachable. This scenario is rather common, so it would not be appropriate for the client system to act as it would if a local file system were lost. Rather, the system can either terminate all operations to the lost server or delay operations until the server is again reachable. These failure semantics are defined and implemented as part of the remote-file-system protocol. Termination of all operations can result in users' losing data—and patience. Thus, most DFS protocols either enforce or allow delaying of file-system operations to remote hosts, with the hope that the remote host will become available again.

To implement this kind of recovery from failure, some kind of **state information** may be maintained on both the client and the server. If both server and client maintain knowledge of their current activities and open files, then they can seamlessly recover from a failure. In the situation where the server crashes but must recognize that it has remotely mounted exported file systems and opened files, NFS Version 3 takes a simple approach, implementing a **stateless** DFS. In essence, it assumes that a client request for a file read or write would not have occurred unless the file system had been remotely mounted and the file had been previously open. The NFS protocol carries all the information needed to locate the appropriate file and perform the requested operation. Similarly, it does not track which clients have the exported volumes mounted, again assuming that if a request comes in, it must be legitimate. While this stateless approach makes NFS resilient and rather easy to implement, it also makes it unsecure. For example, forged read or write requests could be allowed by an NFS server. These issues are addressed in the industry standard NFS Version 4, in which NFS is made stateful to improve its security, performance, and functionality.

## 15.7 Consistency Semantics

Consistency semantics represent an important criterion for evaluating any file system that supports file sharing. These semantics specify how multiple users of a system are to access a shared file simultaneously. In particular, they specify when modifications of data by one user will be observable by other users. These semantics are typically implemented as code with the file system.

Consistency semantics are directly related to the process synchronization algorithms of Chapter 6. However, the complex algorithms of that chapter tend not to be implemented in the case of file I/O because of the great latencies and slow transfer rates of disks and networks. For example, performing an atomic transaction to a remote disk could involve several network communications, several disk reads and writes, or both. Systems that attempt such a full set of functionalities tend to perform poorly. A successful implementation of complex sharing semantics can be found in the Andrew file system.

For the following discussion, we assume that a series of file accesses (that is, reads and writes) attempted by a user to the same file is always enclosed between the `open()` and `close()` operations. The series of accesses between the `open()` and `close()` operations makes up a **file session**. To illustrate the concept, we sketch several prominent examples of consistency semantics.

### 15.7.1 UNIX Semantics

The UNIX file system (Chapter 19) uses the following consistency semantics:

- Writes to an open file by a user are visible immediately to other users who have this file open.
- One mode of sharing allows users to share the pointer of current location into the file. Thus, the advancing of the pointer by one user affects all sharing users. Here, a file has a single image that interleaves all accesses, regardless of their origin.

In the UNIX semantics, a file is associated with a single physical image that is accessed as an exclusive resource. Contention for this single image causes delays in user processes.

### 15.7.2 Session Semantics

The Andrew file system (OpenAFS) uses the following consistency semantics:

- Writes to an open file by a user are not visible immediately to other users that have the same file open.
- Once a file is closed, the changes made to it are visible only in sessions starting later. Already open instances of the file do not reflect these changes.

According to these semantics, a file may be associated temporarily with several (possibly different) images at the same time. Consequently, multiple users are allowed to perform both read and write accesses concurrently on their images of the file, without delay. Almost no constraints are enforced on scheduling accesses.

### 15.7.3 Immutable-Shared-Files Semantics

A unique approach is that of **immutable shared files**. Once a file is declared as shared by its creator, it cannot be modified. An immutable file has two key properties: its name may not be reused, and its contents may not be altered. Thus, the name of an immutable file signifies that the contents of the file are fixed. The implementation of these semantics in a distributed system (Chapter 19) is simple, because the sharing is disciplined (read-only).

## 15.8 NFS

Network file systems are commonplace. They are typically integrated with the overall directory structure and interface of the client system. NFS is a good example of a widely used, well implemented client–server network file system. Here, we use it as an example to explore the implementation details of network file systems.

NFS is both an implementation and a specification of a software system for accessing remote files across LANs (or even WANs). NFS is part of ONC+, which most UNIX vendors and some PC operating systems support. The implementation described here is part of the Solaris operating system, which is a modified version of UNIX SVR4. It uses either the TCP or UDP/IP protocol (depending on the interconnecting network). The specification and the implementation are intertwined in our description of NFS. Whenever detail is needed, we refer to the Solaris implementation; whenever the description is general, it applies to the specification also.

There are multiple versions of NFS, with the latest being Version 4. Here, we describe Version 3, which is the version most commonly deployed.

### 15.8.1 Overview

NFS views a set of interconnected workstations as a set of independent machines with independent file systems. The goal is to allow some degree of sharing among these file systems (on explicit request) in a transparent manner. Sharing is based on a client–server relationship. A machine may be, and often is, both a client and a server. Sharing is allowed between any pair of machines. To ensure machine independence, sharing of a remote file system affects only the client machine and no other machine.

So that a remote directory will be accessible in a transparent manner from a particular machine—say, from ***M1***—a client of that machine must first carry out a mount operation. The semantics of the operation involve mounting a remote directory over a directory of a local file system. Once the mount operation is completed, the mounted directory looks like an integral subtree of the local file system, replacing the subtree descending from the local directory. The local directory becomes the name of the root of the newly mounted directory. Specification of the remote directory as an argument for the mount operation is not done transparently; the location (or host name) of the remote directory has to be provided. However, from then on, users on machine ***M1*** can access files in the remote directory in a totally transparent manner.

To illustrate file mounting, consider the file system depicted in Figure 15.6, where the triangles represent subtrees of directories that are of interest. The figure shows three independent file systems of machines named ***U***, ***S1***, and ***S2***. At this point, on each machine, only the local files can be accessed. Figure 15.7(a) shows the effects of mounting `S1:/usr/shared` over `U:/usr/local`. This figure depicts the view users on ***U*** have of their file system. After the mount is complete, they can access any file within the `dir1` directory using the prefix `/usr/local/dir1`. The original directory `/usr/local` on that machine is no longer visible.

Subject to access-rights accreditation, any file system, or any directory within a file system, can be mounted remotely on top of any local directory.

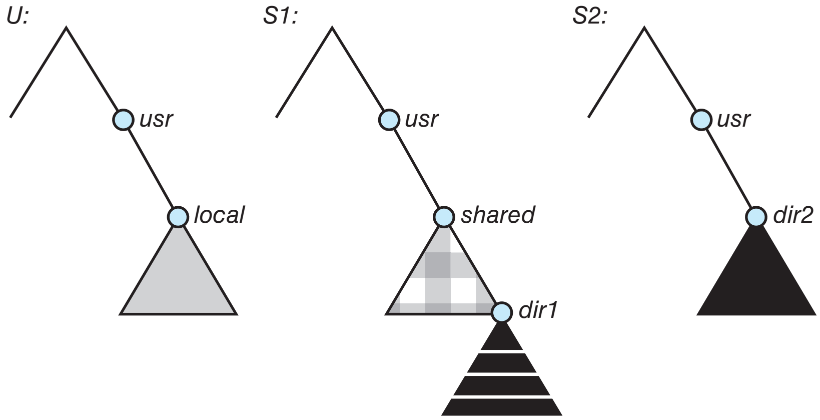


**Figure 15.6** Three independent file systems.

Diskless workstations can even mount their own roots from servers. Cascading mounts are also permitted in some NFS implementations. That is, a file system can be mounted over another file system that is remotely mounted, not local. A machine is affected by only those mounts that it has itself invoked. Mounting a remote file system does not give the client access to other file systems that were, by chance, mounted over the former file system. Thus, the mount mechanism does not exhibit a transitivity property.

In Figure 15.7(b), we illustrate cascading mounts. The figure shows the result of mounting `S2:/usr/dir2` over `U:/usr/local/dir1`, which is already remotely mounted from ***S1***. Users can access files within `dir2` on ***U*** using the prefix `/usr/local/dir1`. If a shared file system is mounted over a user's home directories on all machines in a network, the user can log into any workstation and get his or her home environment. This property permits user mobility.

One of the design goals of NFS was to operate in a heterogeneous environment of different machines, operating systems, and network architectures. The

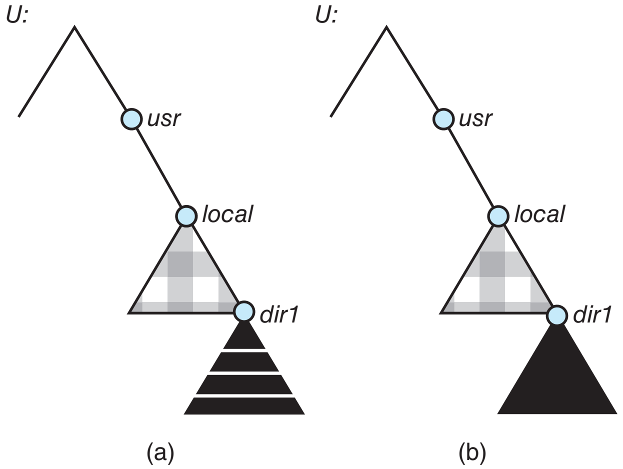


**Figure 15.7** Mounting in NFS. (a) Mounts. (b) Cascading mounts.

NFS specification is independent of these media. This independence is achieved through the use of RPC primitives built on top of an external data representation (XDR) protocol used between two implementation-independent interfaces. Hence, if the system's heterogeneous machines and file systems are properly interfaced to NFS, file systems of different types can be mounted both locally and remotely.

The NFS specification distinguishes between the services provided by a mount mechanism and the actual remote-file-access services. Accordingly, two separate protocols are specified for these services: a mount protocol and a protocol for remote file accesses, the **NFS protocol**. The protocols are specified as sets of RPCs. These RPCs are the building blocks used to implement transparent remote file access.

### 15.8.2 The Mount Protocol

The **mount protocol** establishes the initial logical connection between a server and a client. In Solaris, each machine has a server process, outside the kernel, performing the protocol functions.

A mount operation includes the name of the remote directory to be mounted and the name of the server machine storing it. The mount request is mapped to the corresponding RPC and is forwarded to the mount server running on the specific server machine. The server maintains an **export list** that specifies local file systems that it exports for mounting, along with names of machines that are permitted to mount them. (In Solaris, this list is the `/etc/dfs/dfstab`, which can be edited only by a superuser.) The specification can also include access rights, such as read only. To simplify the maintenance of export lists and mount tables, a distributed naming scheme can be used to hold this information and make it available to appropriate clients.

Recall that any directory within an exported file system can be mounted remotely by an accredited machine. A component unit is such a directory. When the server receives a mount request that conforms to its export list, it returns to the client a file handle that serves as the key for further accesses to files within the mounted file system. The file handle contains all the information that the server needs to distinguish an individual file it stores. In UNIX terms, the file handle consists of a file-system identifier and an inode number to identify the exact mounted directory within the exported file system.

The server also maintains a list of the client machines and the corresponding currently mounted directories. This list is used mainly for administrative purposes—for instance, for notifying all clients that the server is going down. Only through addition and deletion of entries in this list can the server state be affected by the mount protocol.

Usually, a system has a static mounting preconfiguration that is established at boot time (`/etc/vfstab` in Solaris); however, this layout can be modified. In addition to the actual mount procedure, the mount protocol includes several other procedures, such as unmount and return export list.

### 15.8.3 The NFS Protocol

The NFS protocol provides a set of RPCs for remote file operations. The procedures support the following operations:

- Searching for a file within a directory
- Reading a set of directory entries
- Manipulating links and directories
- Accessing file attributes
- Reading and writing files

These procedures can be invoked only after a file handle for the remotely mounted directory has been established.

The omission of open and close operations is intentional. A prominent feature of NFS servers is that they are stateless. Servers do not maintain information about their clients from one access to another. No parallels to UNIX's open-files table or file structures exist on the server side. Consequently, each request has to provide a full set of arguments, including a unique file identifier and an absolute offset inside the file for the appropriate operations. The resulting design is robust; no special measures need be taken to recover a server after a crash. File operations must be idempotent for this purpose—that is, the same operation performed multiple times must have the same effect as if it had only been performed once. To achieve idempotence, every NFS request has a sequence number, allowing the server to determine if a request has been duplicated or if any are missing.

Maintaining the list of clients that we mentioned seems to violate the statelessness of the server. However, this list is not essential for the correct operation of the client or the server, and hence it does not need to be restored after a server crash. Consequently, it may include inconsistent data and is treated as only a hint.

A further implication of the stateless-server philosophy and a result of the synchrony of an RPC is that modified data (including indirection and status blocks) must be committed to the server's disk before results are returned to the client. That is, a client can cache write blocks, but when it flushes them to the server, it assumes that they have reached the server's disks. The server must write all NFS data synchronously. Thus, a server crash and recovery will be invisible to a client; all blocks that the server is managing for the client will be intact. The resulting performance penalty can be large, because the advantages of caching are lost. Performance can be increased by using storage with its own nonvolatile cache (usually battery-backed-up memory). The disk controller acknowledges the disk write when the write is stored in the nonvolatile cache. In essence, the host sees a very fast synchronous write. These blocks remain intact even after a system crash and are written from this stable storage to disk periodically.

A single NFS write procedure call is guaranteed to be atomic and is not intermixed with other write calls to the same file. The NFS protocol, however, does not provide concurrency-control mechanisms. A `write()` system call may be broken down into several RPC writes, because each NFS write or read call can contain up to 8 KB of data and UDP packets are limited to 1,500 bytes. As a result, two users writing to the same remote file may get their data intermixed. The claim is that, because lock management is inherently stateful, a service outside the NFS should provide locking (and Solaris does). Users are advised to coordinate access to shared files using mechanisms outside the scope of NFS.

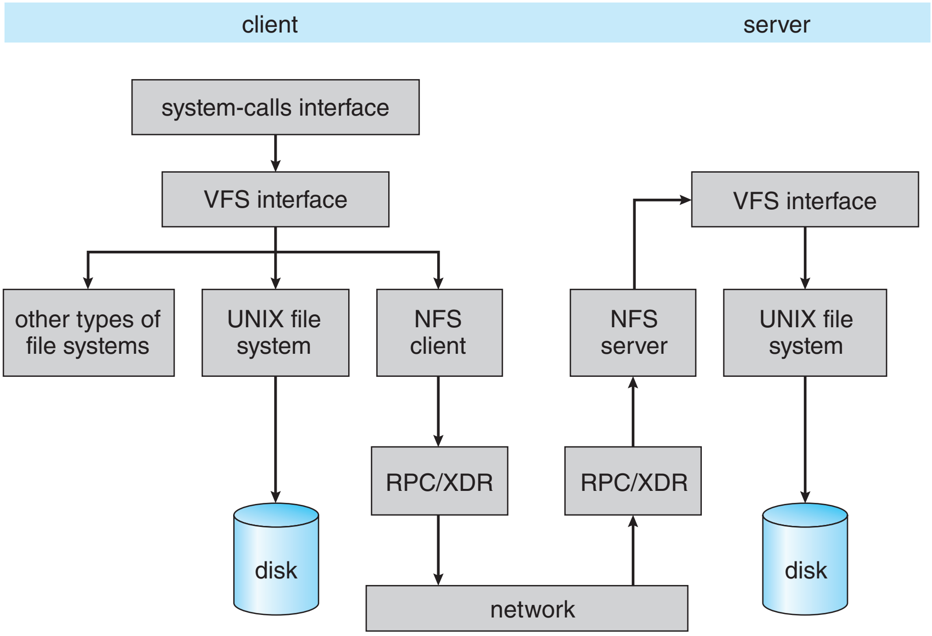


**Figure 15.8** Schematic view of the NFS architecture.

NFS is integrated into the operating system via a VFS. As an illustration of the architecture, let's trace how an operation on an already-open remote file is handled (follow the example in Figure 15.8). The client initiates the operation with a regular system call. The operating-system layer maps this call to a VFS operation on the appropriate vnode. The VFS layer identifies the file as a remote one and invokes the appropriate NFS procedure. An RPC call is made to the NFS service layer at the remote server. This call is reinjected to the VFS layer on the remote system, which finds that it is local and invokes the appropriate file-system operation. This path is retraced to return the result. An advantage of this architecture is that the client and the server are identical; thus, a machine may be a client, or a server, or both. The actual service on each server is performed by kernel threads.

### 15.8.4 Path-Name Translation

**Path-name translation** in NFS involves the parsing of a path name such as `/usr/local/dir1/file.txt` into separate directory entries, or components: (1) `usr`, (2) `local`, and (3) `dir1`. Path-name translation is done by breaking the path into component names and performing a separate NFS `lookup call` for every pair of component name and directory vnode. Once a mount point is crossed, every component lookup causes a separate RPC to the server. This expensive path-name-traversal scheme is needed, since the layout of each client's logical name space is unique, dictated by the mounts the client has performed. It would be much more efficient to hand a server a path name and receive a target vnode once a mount point is encountered. At any point, however, there might be another mount point for the particular client of which the stateless server is unaware.

So that lookup is fast, a directory-name-lookup cache on the client side holds the vnodes for remote directory names. This cache speeds up references to files with the same initial path name. The directory cache is discarded when attributes returned from the server do not match the attributes of the cached vnode.

Recall that some implementations of NFS allow mounting a remote file system on top of another already-mounted remote file system (a cascading mount). When a client has a cascading mount, more than one server can be involved in a path-name traversal. However, when a client does a lookup on a directory on which the server has mounted a file system, the client sees the underlying directory instead of the mounted directory.

### 15.8.5 Remote Operations

With the exception of opening and closing files, there is an almost one-to-one correspondence between the regular UNIX system calls for file operations and the NFS protocol RPCs. Thus, a remote file operation can be translated directly to the corresponding RPC. Conceptually, NFS adheres to the remote-service paradigm; but in practice, buffering and caching techniques are employed for the sake of performance. No direct correspondence exists between a remote operation and an RPC. Instead, file blocks and file attributes are fetched by the RPCs and are cached locally. Future remote operations use the cached data, subject to consistency constraints.

There are two caches: the file-attribute (inode-information) cache and the file-blocks cache. When a file is opened, the kernel checks with the remote server to determine whether to fetch or revalidate the cached attributes. The cached file blocks are used only if the corresponding cached attributes are up to date. The attribute cache is updated whenever new attributes arrive from the server. Cached attributes are, by default, discarded after 60 seconds. Both read-ahead and delayed-write techniques are used between the server and the client. Clients do not free delayed-write blocks until the server confirms that the data have been written to disk. Delayed-write is retained even when a file is opened concurrently, in conflicting modes. Hence, UNIX semantics (Section 15.7.1) are not preserved.

Tuning the system for performance makes it difficult to characterize the consistency semantics of NFS. New files created on a machine may not be visible elsewhere for 30 seconds. Furthermore, writes to a file at one site may or may not be visible at other sites that have this file open for reading. New opens of a file observe only the changes that have already been flushed to the server. Thus, NFS provides neither strict emulation of UNIX semantics nor the session semantics of Andrew (Section 15.7.2). In spite of these drawbacks, the utility and good performance of the mechanism make it the most widely used multi-vendor-distributed system in operation.

## 15.9 Summary

- General-purpose operating systems provide many file-system types, from special-purpose through general.

- Volumes containing file systems can be mounted into the computer's file-system space.
- Depending on the operating system, the file-system space is seamless (mounted file systems integrated into the directory structure) or distinct (each mounted file system having its own designation).
- At least one file system must be bootable for the system to be able to start —that is, it must contain an operating system. The boot loader is run first; it is a simple program that is able to find the kernel in the file system, load it, and start its execution. Systems can contain multiple bootable partitions, letting the administrator choose which to run at boot time.
- Most systems are multi-user and thus must provide a method for file sharing and file protection. Frequently, files and directories include metadata, such as owner, user, and group access permissions.
- Mass storage partitions are used either for raw block I/O or for file systems. Each file system resides in a volume, which can be composed of one partition or multiple partitions working together via a volume manager.
- To simplify implementation of multiple file systems, an operating system can use a layered approach, with a virtual file-system interface making access to possibly dissimilar file systems seamless.
- Remote file systems can be implemented simply by using a program such as `ftp` or the web servers and clients in the World Wide Web, or with more functionality via a client–server model. Mount requests and user IDs must be authenticated to prevent unapproved access.
- Client–server facilities do not natively share information, but a distributed information system such as DNS can be used to allow such sharing, providing a unified user name space, password management, and system identification. For example, Microsoft CIFS uses active directory, which employs a version of the Kerberos network authentication protocol to provide a full set of naming and authentication services among the computers in a network.
- Once file sharing is possible, a consistency semantics model must be chosen and implemented to moderate multiple concurrent access to the same file. Semantics models include UNIX, session, and immutable-shared-files semantics.
- NFS is an example of a remote file system, providing clients with seamless access to directories, files, and even entire file systems. A full-featured remote file system includes a communication protocol with remote operations and path-name translation.

## Practice Exercises

**15.1** Explain how the VFS layer allows an operating system to support multiple types of file systems easily.

**15.2** Why have more than one file system type on a given system?

**15.3** On a Unix or Linux system that implements the procfs file system, determine how to use the procfs interface to explore the process name space. What aspects of processes can be viewed via this interface? How would the same information be gathered on a system lacking the procfs file system?

**15.4** Why do some systems integrate mounted file systems into the root file system naming structure, while others use a separate naming method for mounted file systems?

**15.5** Given a remote file access facility such as `ftp`, why were remote file systems like NFS created?

## Further Reading

The internals of the BSD UNIX system are covered in full in [McKusick et al. (2015)]. Details concerning file systems for Linux can be found in [Love (2010)].

The network file system (NFS) is discussed in [Callaghan (2000)]. NFS Version 4 is a standard described at http://www.ietf.org/rfc/rfc3530.txt. [Ousterhout (1991)] discusses the role of distributed state in networked file systems. NFS and the UNIX file system (UFS) are described in [Mauro and McDougall (2007)].

The Kerberos network authentication protocol is explored in https://web.mit.edu/kerberos/.

## Bibliography

**[Callaghan (2000)]** B. Callaghan, *NFS Illustrated*, Addison-Wesley (2000).

**[Love (2010)]** R. Love, *Linux Kernel Development*, Third Edition, Developer's Library (2010).

**[Mauro and McDougall (2007)]** J. Mauro and R. McDougall, *Solaris Internals: Core Kernel Architecture*, Prentice Hall (2007).

**[McKusick et al. (2015)]** M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson, *The Design and Implementation of the FreeBSD UNIX Operating System–Second Edition*, Pearson (2015).

**[Ousterhout (1991)]** J. Ousterhout. "The Role of Distributed State". *In CMU Computer Science: a 25th Anniversary Commemorative*, R. F. Rashid, Ed., Addison-Wesley (1991).

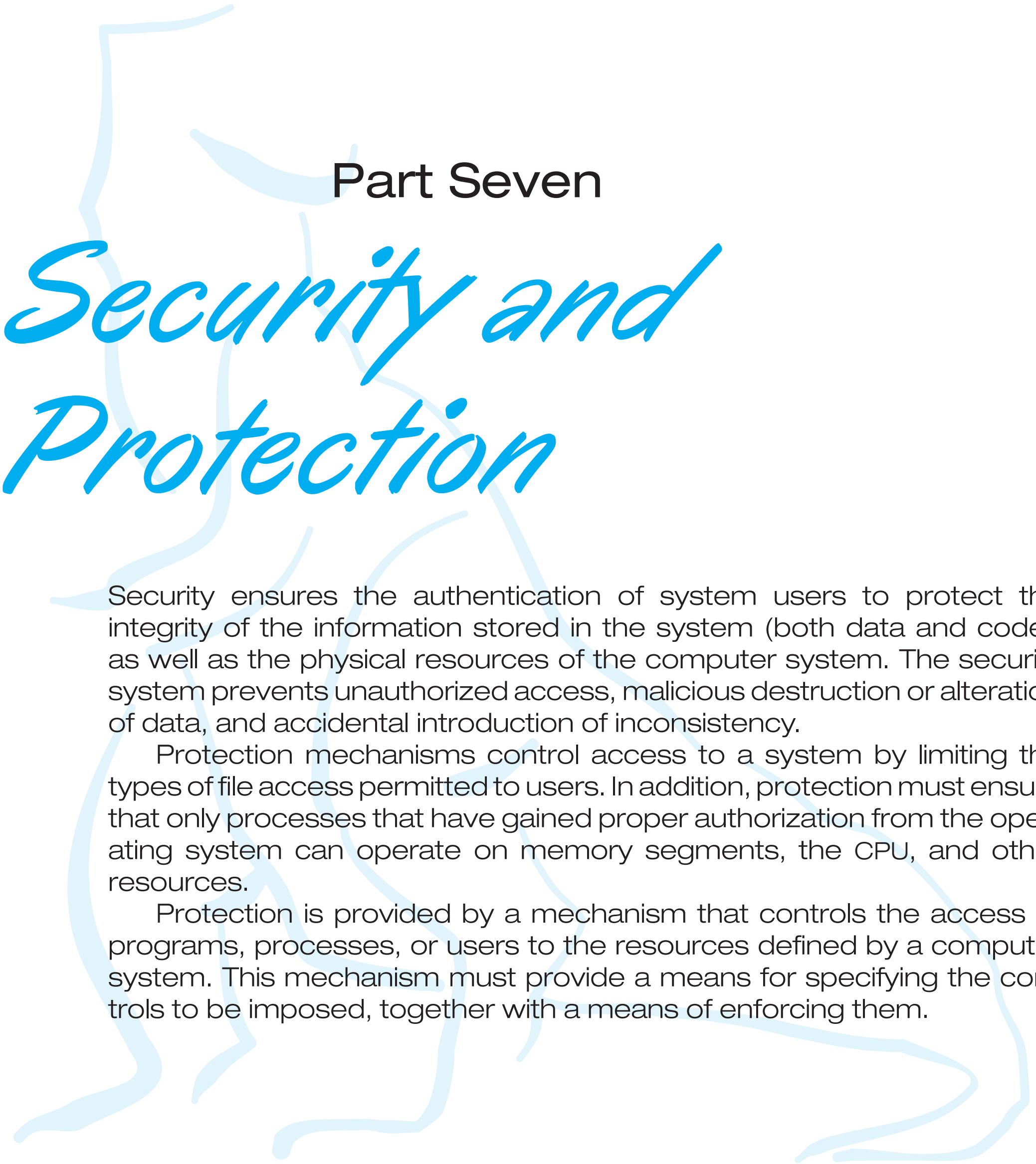
# Chapter 15 Exercises

**15.6** Assume that in a particular augmentation of a remote-file-access protocol, each client maintains a name cache that caches translations from file names to corresponding file handles. What issues should we take into account in implementing the name cache?

**15.7** Given a mounted file system with write operations underway, and a system crash or power loss, what must be done before the file system is remounted if: (a) The file system is not log-structured? (b) The file system is log-structured?

**15.8** Why do operating systems mount the root file system automatically at boot time?

**15.9** Why do operating systems require file systems other than root to be mounted?



## Part Seven

# Security and Protection

Security ensures the authentication of system users to protect the integrity of the information stored in the system (both data and code), as well as the physical resources of the computer system. The security system prevents unauthorized access, malicious destruction or alteration of data, and accidental introduction of inconsistency.

Protection mechanisms control access to a system by limiting the types of file access permitted to users. In addition, protection must ensure that only processes that have gained proper authorization from the operating system can operate on memory segments, the CPU, and other resources.

Protection is provided by a mechanism that controls the access of programs, processes, or users to the resources defined by a computer system. This mechanism must provide a means for specifying the controls to be imposed, together with a means of enforcing them.

# CHAPTER 16

# Security

Both protection and security are vital to computer systems. We distinguish between these two concepts in the following way: Security is a measure of confidence that the integrity of a system and its data will be preserved. Protection is the set of mechanisms that control the access of processes and users to the resources defined by a computer system. We focus on security in this chapter and address protection in Chapter 17.

Security involves guarding computer resources against unauthorized access, malicious destruction or alteration, and accidental introduction of inconsistency. Computer resources include the information stored in the system (both data and code), as well as the CPU, memory, secondary storage, tertiary storage, and networking that compose the computer facility. In this chapter, we start by examining ways in which resources may be accidentally or purposely misused. We then explore a key security enabler—cryptography. Finally, we look at mechanisms to guard against or detect attacks.

## CHAPTER OBJECTIVES

- Discuss security threats and attacks.
- Explain the fundamentals of encryption, authentication, and hashing.
- Examine the uses of cryptography in computing.
- Describe various countermeasures to security attacks.

## 16.1 The Security Problem

In many applications, ensuring the security of the computer system is worth considerable effort. Large commercial systems containing payroll or other financial data are inviting targets to thieves. Systems that contain data pertaining to corporate operations may be of interest to unscrupulous competitors. Furthermore, loss of such data, whether by accident or fraud, can seriously impair the ability of the corporation to function. Even raw computing resources are attractive to attackers for bitcoin mining, for sending spam, and as a source from which to anonymously attack other systems.

In Chapter 17, we discuss mechanisms that the operating system can provide (with appropriate aid from the hardware) that allow users to protect their resources, including programs and data. These mechanisms work well only as long as the users conform to the intended use of and access to these resources.

We say that a system is **secure** if its resources are used and accessed as intended under all circumstances. Unfortunately, total security cannot be achieved. Nonetheless, we must have mechanisms to make security breaches a rare occurrence, rather than the norm.

Security violations (or misuse) of the system can be categorized as intentional (malicious) or accidental. It is easier to protect against accidental misuse than against malicious misuse. For the most part, protection mechanisms are the core of accident avoidance. The following list includes several forms of accidental and malicious security violations. Note that in our discussion of security, we use the terms **intruder**, **hacker**, and **attacker** for those attempting to breach security. In addition, a **threat** is the potential for a security violation, such as the discovery of a vulnerability, whereas an **attack** is an attempt to break security.

- **Breach of confidentiality**. This type of violation involves unauthorized reading of data (or theft of information). Typically, a breach of confidentiality is the goal of an intruder. Capturing secret data from a system or a data stream, such as credit-card information or identity information for identity theft, or unreleased movies or scripts, can result directly in money for the intruder and embarrassment for the hacked institution.
- **Breach of integrity**. This violation involves unauthorized modification of data. Such attacks can, for example, result in passing of liability to an innocent party or modification of the source code of an important commercial or open-source application.
- **Breach of availability**. This violation involves unauthorized destruction of data. Some attackers would rather wreak havoc and get status or bragging rights than gain financially. Website defacement is a common example of this type of security breach.
- **Theft of service**. This violation involves unauthorized use of resources. For example, an intruder (or intrusion program) may install a daemon on a system that acts as a file server.
- **Denial of service**. This violation involves preventing legitimate use of the system. **Denial-of-service** (**DOS**) attacks are sometimes accidental. The original Internet worm turned into a DOS attack when a bug failed to delay its rapid spread. We discuss DOS attacks further in Section 16.3.2.

Attackers use several standard methods in their attempts to breach security. The most common is **masquerading**, in which one participant in a communication pretends to be someone else (another host or another person). By masquerading, attackers breach **authentication**, the correctness of identification; they can then gain access that they would not normally be allowed. Another common attack is to replay a captured exchange of data. A **replay attack** consists of the malicious or fraudulent repeat of a valid data transmission. Sometimes the replay comprises the entire attack—for example, in a repeat of a request to transfer money. But frequently it is done along with **message**

**modification**, in which the attacker changes data in a communication without the sender's knowledge. Consider the damage that could be done if a request for authentication had a legitimate user's information replaced with an unauthorized user's. Yet another kind of attack is the **man-in-the-middle attack**, in which an attacker sits in the data flow of a communication, masquerading as the sender to the receiver, and vice versa. In a network communication, a man-in-the-middle attack may be preceded by a **session hijacking**, in which an active communication session is intercepted.

Another broad class of attacks is aimed at **privilege escalation**. Every system assigns privileges to users, even if there is just one user and that user is the administrator. Generally, the system includes several sets of privileges, one for each user account and some for the system. Frequently, privileges are also assigned to nonusers of the system (such as users from across the Internet accessing a web page without logging in or anonymous users of services such as file transfer). Even a sender of email to a remote system can be considered to have privileges—the privilege of sending an email to a receiving user on that system. Privilege escalation gives attackers more privileges than they are supposed to have. For example, an email containing a script or macro that is executed exceeds the email sender's privileges. Masquerading and message modification, mentioned above, are often done to escalate privileges. There are many more examples, as this is a very common type of attack. Indeed, it is difficult to detect and prevent all of the various attacks in this category.

As we have already suggested, absolute protection of the system from malicious abuse is not possible, but the cost to the perpetrator can be made sufficiently high to deter most intruders. In some cases, such as a denial-of-service attack, it is preferable to prevent the attack but sufficient to detect it so that countermeasures can be taken (such as up-stream filtering or adding resources such that the attack is not denying services to legitimate users).

To protect a system, we must take security measures at four levels:

1. **Physical**. The site or sites containing the computer systems must be physically secured against entry by intruders. Both the machine rooms and the terminals or computers that have access to the target machines must be secured, for example by limiting access to the building they reside in, or locking them to the desk on which they sit.

2. **Network**. Most contemporary computer systems—from servers to mobile devices to Internet of Things (IoT) devices—are networked. Networking provides a means for the system to access external resources but also provides a potential vector for unauthorized access to the system itself.

   Further, computer data in modern systems frequently travel over private leased lines, shared lines like the Internet, wireless connections, and dial-up lines. Intercepting these data can be just as harmful as breaking into a computer, and interruption of communications can constitute a remote denial-of-service attack, diminishing users' use of and trust in the system.

3. **Operating system**. The operating system and its built-in set of applications and services comprise a huge code base that may harbor many vulnerabilities. Insecure default settings, misconfigurations, and security

bugs are only a few potential problems. Operating systems must thus be kept up to date (via continuous patching) and “hardened”—configured and modified to decrease the attack surface and avoid penetration. The **attack surface** is the set of points at which an attacker can try to break into the system.

4. **Application**. Third-party applications may also pose risks, especially if they possess significant privileges. Some applications are inherently malicious, but even benign applications may contain security bugs. Due to the vast number of third-party applications and their disparate code bases, it is virtually impossible to ensure that all such applications are secure.

This four-layered security model is shown in Figure 16.1.

The four-layer model of security is like a chain made of links: a vulnerability in any of its layers can lead to full system compromise. In that respect, the old adage that security is only as strong as its weakest link holds true.

Another factor that cannot be overlooked is the human one. Authorization must be performed carefully to ensure that only allowed, trusted users have access to the system. Even authorized users, however, may be malicious or may be “encouraged” to let others use their access—whether willingly or when duped through **social engineering**, which uses deception to persuade people to give up confidential information. One type of social-engineering attack is **phishing**, in which a legitimate-looking e-mail or web page misleads a user into entering confidential information. Sometimes, all it takes is a click of a link on a browser page or in an email to inadvertently download a malicious payload, compromising system security on the user’s computer. Usually that PC is not the end target, but rather some more valuable resource. From that compromised system, attacks on other systems on the LAN or other users ensue.

So far, we’ve seen that all four factors in the four-level model, plus the human factor, must be taken into account if security is to be maintained. Furthermore, the system must provide protection (discussed in great detail in Chapter 17) to allow the implementation of security features. Without the ability to authorize users and processes to control their access, and to log their activities, it would be impossible for an operating system to implement security measures or to run securely. Hardware protection features are needed to support an overall protection scheme. For example, a system without memory

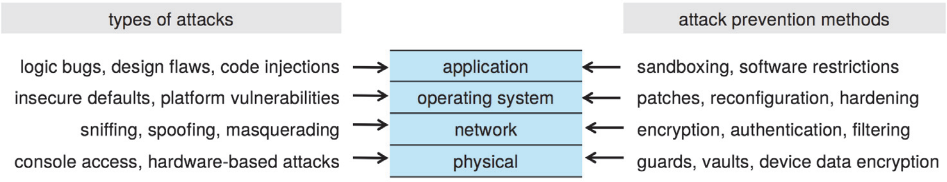

types of attacks
attack prevention methods
logic bugs, design flaws, code injections
application
sandboxing, software restrictions
insecure defaults, platform vulnerabilities
operating system
patches, reconfiguration, hardening
sniffing, spoofing, masquerading
network
encryption, authentication, filtering
console access, hardware-based attacks
physical
guards, vaults, device data encryption

**Figure 16.1** The four-layered model of security.

protection cannot be secure. New hardware features are allowing systems to be made more secure, as we shall discuss.

Unfortunately, little in security is straightforward. As intruders exploit security vulnerabilities, security countermeasures are created and deployed. This causes intruders to become more sophisticated in their attacks. For example, spyware can provide a conduit for spam through innocent systems (we discuss this practice in Section 16.2), which in turn can deliver phishing attacks to other targets. This cat-and-mouse game is likely to continue, with more security tools needed to block the escalating intruder techniques and activities.

In the remainder of this chapter, we address security at the network and operating-system levels. Security at the application, physical and human levels, although important, is for the most part beyond the scope of this text. Security within the operating system and between operating systems is implemented in several ways, ranging from passwords for authentication through guarding against viruses to detecting intrusions. We start with an exploration of security threats.

## 16.2 Program Threats

Processes, along with the kernel, are the only means of accomplishing work on a computer. Therefore, writing a program that creates a breach of security, or causing a normal process to change its behavior and create a breach, is a common goal of attackers. In fact, even most nonprogram security events have as their goal causing a program threat. For example, while it is useful to log in to a system without authorization, it is quite a lot more useful to leave behind a back-door daemon or **Remote Access Tool** (**RAT**) that provides information or allows easy access even if the original exploit is blocked. In this section, we describe common methods by which programs cause security breaches. Note that there is considerable variation in the naming conventions for security holes and that we use the most common or descriptive terms.

### 16.2.1 Malware

**Malware** is software designed to exploit, disable or damage computer systems. There are many ways to perform such activities, and we explore the major variations in this section.

Many systems have mechanisms for allowing programs written by a user to be executed by other users. If these programs are executed in a domain that provides the access rights of the executing user, the other users may misuse these rights. A program that acts in a clandestine or malicious manner, rather than simply performing its stated function, is called a **Trojan horse**. If the program is executed in another domain, it can escalate privileges. As an example, consider a mobile app that purports to provide some benign functionality—say, a flashlight app—but that meanwhile surreptitiously accesses the user's contacts or messages and smuggles them to some remote server.

A classic variation of the Trojan horse is a “Trojan mule” program that emulates a login program. An unsuspecting user starts to log in at a terminal, computer, or web page and notices that she has apparently mistyped her password. She tries again and is successful. What has happened is that her authentication key and password have been stolen by the login emulator, which was left running on the computer by the attacker or reached via a bad URL. The emulator stored away the password, printed out a login error message, and exited; the user was then provided with a genuine login prompt. This type of attack can be defeated by having the operating system print a usage message at the end of an interactive session, by requiring a nontrappable key sequence to get to the login prompt, such as the `control-alt-delete` combination used by all modern Windows operating systems, or by the user ensuring the URL is the right, valid one.

Another variation on the Trojan horse is **spyware**. Spyware sometimes accompanies a program that the user has chosen to install. Most frequently, it comes along with freeware or shareware programs, but sometimes it is included with commercial software. Spyware may download ads to display on the user's system, create pop-up browser windows when certain sites are visited, or capture information from the user's system and return it to a central site. The installation of an innocuous-seeming program on a Windows system could result in the loading of a spyware daemon. The spyware could contact a central site, be given a message and a list of recipient addresses, and deliver a spam message to those users from the Windows machine. This process would continue until the user discovered the spyware. Frequently, the spyware is not discovered. In 2010, it was estimated that 90 percent of spam was being delivered by this method. This theft of service is not even considered a crime in most countries!

A fairly recent and unwelcome development is a class of malware that doesn't steal information. **Ransomware** encrypts some or all of the information on the target computer and renders it inaccessible to the owner. The information itself has little value to the attacker but lots of value to the owner. The idea is to force the owner to pay money (the ransom) to get the decryption key needed to decrypt the data. As with other dealings with criminals, of course, payment of the ransom does not guarantee return of access.

Trojans and other malware especially thrive in cases where there is a violation of the **principle of least privilege**. This commonly occurs when the operating system allows by default more privileges than a normal user needs or when the user runs by default as an administrator (as was true in all Windows operating systems up to Windows 7). In such cases, the operating system's own immune system—permissions and protections of various kinds—cannot "kick in," so the malware can persist and survive across reboot, as well as extend its reach both locally and over the network.

Violating the principle of least privilege is a case of poor operating-system design decision making. An operating system (and, indeed, software in general) should allow fine-grained control of access and security, so that only the privileges needed to perform a task are available during the task's execution. The control feature must also be easy to manage and understand. Inconvenient, inadequate, and misunderstood security measures are bound to be circumvented, causing an overall weakening of the security they were designed to implement.

In yet another form of malware, the designer of a program or system leaves a hole in the software that only she is capable of using. This type of security breach, a **trap door** (or **back door**), was shown in the movie ***War Games.*** For

### *THE PRINCIPLE OF LEAST PRIVILEGE*

“The principle of least privilege. Every program and every privileged user of the system should operate using the least amount of privilege necessary to complete the job. The purpose of this principle is to reduce the number of potential interactions among privileged programs to the minimum necessary to operate correctly, so that one may develop confidence that unintentional, unwanted, or improper uses of privilege do not occur.”—Jerome H. Saltzer, describing a design principle of the Multics operating system in 1974: https://pdfs.semanticscholar.org/1c8d/06510ad449ad24fbdd164f8008cc730cab47.pdf.

instance, the code might check for a specific user ID or password, and it might circumvent normal security procedures when it receives that ID or password. Programmers have used the trap-door method to embezzle from banks by including rounding errors in their code and having the occasional half-cent credited to their accounts. This account crediting can add up to a large amount of money, considering the number of transactions that a large bank executes.

A trap door may be set to operate only under a specific set of logic conditions, in which case it is referred to as a **logic bomb**. Back doors of this type are especially difficult to detect, as they may remain dormant for a long time, possibly years, before being detected—usually after the damage has been done. For example, one network administrator had a destructive reconfiguration of his company’s network execute when his program detected that he was no longer employed at the company.

A clever trap door could be included in a compiler. The compiler could generate standard object code as well as a trap door, regardless of the source code being compiled. This activity is particularly nefarious, since a search of the source code of the program will not reveal any problems. Only reverse engineering of the code of the compiler itself would reveal this trap door. This type of attack can also be performed by patching the compiler or compile-time libraries after the fact. Indeed, in 2015, malware that targets Apple’s XCode compiler suite (dubbed “XCodeGhost”) affected many software developers who used compromised versions of XCode not downloaded directly from Apple.

Trap doors pose a difficult problem because, to detect them, we have to analyze all the source code for all components of a system. Given that software systems may consist of millions of lines of code, this analysis is not done frequently, and frequently it is not done at all! A software development methodology that can help counter this type of security hole is **code review**. In code review, the developer who wrote the code submits it to the code base, and one or more developers review the code and approve it or provide comments. Once a defined set of reviewers approve the code (sometimes after comments are addressed and the code is resubmitted and re-reviewed), the code is admitted into the code base and then compiled, debugged, and finally released for use. Many good software developers use development version control systems that provide tools for code review—for example, `git` (https://github.com/git/). Note, too, that there are automatic code-review and

```
#include <stdio.h>
#define BUFFER_SIZE 0

int main(int argc, char *argv[])
{
   int j = 0;
   char buffer[BUFFER_SIZE];
   int k = 0;
   if (argc < 2) {return -1;}

   strcpy(buffer,argv[1]);
   printf("K is %d, J is %d, buffer is %s\n", j,k,buffer);
   return 0;
   }
}
```

**Figure 16.2** C program with buffer-overflow condition.

code-scanning tools designed to find flaws, including security flaws, but generally good programmers are the best code reviewers.

For those not involved in developing the code, code review is useful for finding and reporting flaws (or for finding and exploiting them). For most software, source code is not available, making code review much harder for nondevelopers.

### 16.2.2 Code Injection

Most software is not malicious, but it can nonetheless pose serious threats to security due to a **code-injection attack**, in which executable code is added or modified. Even otherwise benign software can harbor vulnerabilities that, if exploited, allow an attacker to take over the program code, subverting its existing code flow or entirely reprogramming it by supplying new code.

Code-injection attacks are nearly always the result of poor or insecure programming paradigms, commonly in low-level languages such as C or C++, which allow direct memory access through pointers. This direct memory access, coupled with the need to carefully decide on sizes of memory buffers and take care not to exceed them, can lead to memory corruption when memory buffers are not properly handled.

As an example, consider the simplest code-injection vector—a buffer overflow. The program in Figure 16.2 illustrates such an overflow, which occurs due to an unbounded copy operation, the call to `strcpy()`. The function copies with no regard to the buffer size in question, halting only when a NULL (\0) byte is encountered. If such a byte occurs before the BUFFER_SIZE is reached, the program behaves as expected. But the copy could easily exceed the buffer size—what then?

The answer is that the outcome of an overflow depends largely on the length of the overflow and the overflowing contents (Figure 16.3). It also varies greatly with the code generated by the compiler, which may be optimized

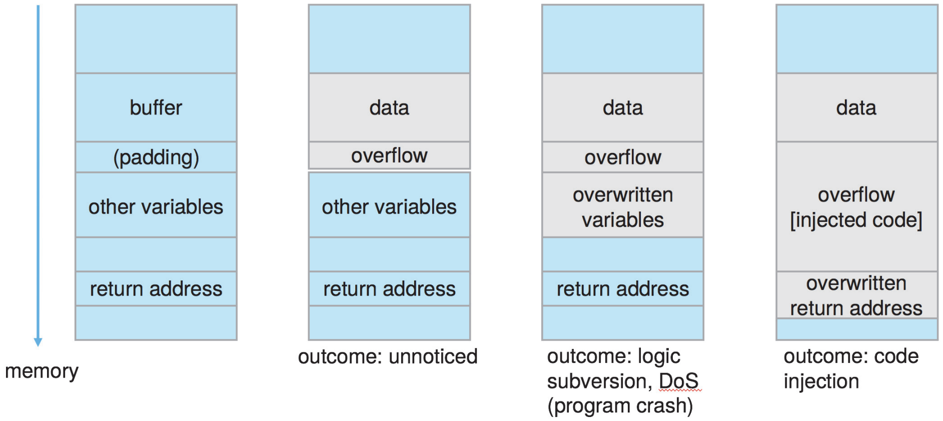

buffer
(padding)
other variables
return address
memory
data
overflow
other variables
return address
outcome: unnoticed
data
overflow
overwritten variables
return address
outcome: logic subversion, DoS (program crash)
data
overflow [injected code]
overwritten return address
outcome: code injection

**Figure 16.3** The possible outcomes of buffer overflows.

in ways that affect the outcome: optimizations often involve adjustments to memory layout (commonly, repositioning or padding variables).

1. If the overflow is very small (only a little more than BUFFER_SIZE), there is a good chance it will go entirely unnoticed. This is because the allocation of BUFFER_SIZE bytes will often be padded to an architecture-specified boundary (commonly 8 or 16 bytes). Padding is unused memory, and therefore an overflow into it, though technically out of bounds, has no ill effect.
2. If the overflow exceeds the padding, the next automatic variable on the stack will be overwritten with the overflowing contents. The outcome here will depend on the exact positioning of the variable and on its semantics (for example, if it is employed in a logical condition that can then be subverted). If uncontrolled, this overflow could lead to a program crash, as an unexpected value in a variable could lead to an uncorrectable error.
3. If the overflow greatly exceeds the padding, all of the current function's stack frame is overwritten. At the very top of the frame is the function's return address, which is accessed when the function returns. The flow of the program is subverted and can be redirected by the attacker to another region of memory, including memory controlled by the attacker (for example, the input buffer itself, or the stack or the heap). The injected code is then executed, allowing the attacker to run arbitrary code as the processes' effective ID.

Note that a careful programmer could have performed bounds checking on the size of `argv[1]` by using the `strncpy()` function rather than `strcpy()`, replacing the line "`strcpy(buffer, argv[1]);`" with "`strncpy(buffer, argv[1], sizeof(buffer)-1);`". Unfortunately, good bounds checking is the exception rather than the norm. `strcpy()` is one of a known class of vulnerable functions, which include `sprintf()`, `gets()`, and other functions with no regard to buffer sizes. But even size-aware variants can harbor vulnerabilities when coupled with arithmetic operations over finite-length integers, which may lead to an integer overflow.

At this point, the dangers inherent in a simple oversight in maintaining a buffer should be clearly evident. Brian Kerningham and Dennis Ritchie (in their book *The C Programming Language*) referred to the possible outcome as "undefined behavior," but perfectly predictable behavior can be coerced by an attacker, as was first demonstrated by the Morris Worm (and documented in RFC1135: https://tools.ietf.org/html/rfc1135). It was not until several years later, however, that an article in issue 49 of *Phrack* magazine ("Smashing the Stack for Fun and Profit" http://phrack.org/issues/49/14.html) introduced the exploitation technique to the masses, unleashing a deluge of exploits.

To achieve code injection, there must first be injectable code. The attacker first writes a short code segment such as the following:

```
void func (void) {

  execvp("/bin/sh", "/bin/sh", NULL); ;
}
```

Using the `execvp()` system call, this code segment creates a shell process. If the program being attacked runs with root permissions, this newly created shell will gain complete access to the system. Of course, the code segment can do anything allowed by the privileges of the attacked process. The code segment is next compiled into its assembly binary opcode form and then transformed into a binary stream. The compiled form is often referred to as shellcode, due to its classic function of spawning a shell, but the term has grown to encompass any type of code, including more advanced code used to add new users to a system, reboot, or even connect over the network and wait for remote instructions (called a "reverse shell"). A shellcode exploit is shown in Figure 16.4. Code that is briefly used, only to redirect execution to some other location, is much like a trampoline, "bouncing" code flow from one spot to another.

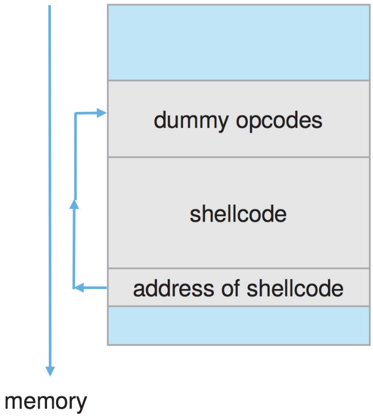


**Figure 16.4** Trampoline to code execution when exploiting a buffer overflow.

There are, in fact, shellcode compilers (the "MetaSploit" project being a notable example), which also take care of such specifics as ensuring that the code is compact and contains no NULL bytes (in case of exploitation via string copy, which would terminate on NULLs). Such a compiler may even mask the shellcode as alphanumeric characters.

If the attacker has managed to overwrite the return address (or any function pointer, such as that of a `VTable`), then all it takes (in the simple case) is to redirect the address to point to the supplied shellcode, which is commonly loaded as part of the user input, through an environment variable, or over some file or network input. Assuming no mitigations exist (as described later), this is enough for the shellcode to execute and the hacker to succeed in the attack. Alignment considerations are often handled by adding a sequence of NOP instructions before the shellcode. The result is known as a NOP-sled, as it causes execution to "slide" down the NOP instructions until the payload is encountered and executed.

This example of a buffer-overflow attack reveals that considerable knowledge and programming skill are needed to recognize exploitable code and then to exploit it. Unfortunately, it does not take great programmers to launch security attacks. Rather, one hacker can determine the bug and then write an exploit. Anyone with rudimentary computer skills and access to the exploit—a so-called **script kiddie**—can then try to launch the attack at target systems.

The buffer-overflow attack is especially pernicious because it can be run between systems and can travel over allowed communication channels. Such attacks can occur within protocols that are expected to be used to communicate with the target machine, and they can therefore be hard to detect and prevent. They can even bypass the security added by firewalls (Section 16.6.6).

Note that buffer overflows are just one of several vectors which can be manipulated for code injection. Overflows can also be exploited when they occur in the heap. Using memory buffers after freeing them, as well as over-freeing them (calling `free()` twice), can also lead to code injection.

### 16.2.3 Viruses and Worms

Another form of program threat is a **virus**. A virus is a fragment of code embedded in a legitimate program. Viruses are self-replicating and are designed to "infect" other programs. They can wreak havoc in a system by modifying or destroying files and causing system crashes and program malfunctions. As with most penetration attacks (direct attacks on a system), viruses are very specific to architectures, operating systems, and applications. Viruses are a particular problem for users of PCs. UNIX and other multiuser operating systems generally are not susceptible to viruses because the executable programs are protected from writing by the operating system. Even if a virus does infect such a program, its powers usually are limited because other aspects of the system are protected.

Viruses are usually borne via spam e-mail and phishing attacks. They can also spread when users download viral programs from Internet file-sharing services or exchange infected disks. A distinction can be made between viruses, which require human activity, and **worms**, which use a network to replicate without any help from humans.

For an example of how a virus "infects" a host, consider Microsoft Office files. These files can contain ***macros*** (or Visual Basic programs) that programs in the Office suite (Word, PowerPoint, and Excel) will execute automatically. Because these programs run under the user's own account, the macros can run largely unconstrained (for example, deleting user files at will). The following code sample shows how simple it is to write a Visual Basic macro that a worm could use to format the hard drive of a Windows computer as soon as the file containing the macro was opened:

```
Sub AutoOpen()
Dim oFS
    Set oFS = CreateObject(”Scripting.FileSystemObject”)
    vs = Shell(”c: command.com /k format c:”,vbHide)
End Sub
```

Commonly, the worm will also e-mail itself to others in the user's contact list.

How do viruses work? Once a virus reaches a target machine, a program known as a **virus dropper** inserts the virus into the system. The virus dropper is usually a Trojan horse, executed for other reasons but installing the virus as its core activity. Once installed, the virus may do any one of a number of things. There are literally thousands of viruses, but they fall into several main categories. Note that many viruses belong to more than one category.

- **File**. A standard file virus infects a system by appending itself to a file. It changes the start of the program so that execution jumps to its code. After it executes, it returns control to the program so that its execution is not noticed. File viruses are sometimes known as parasitic viruses, as they leave no full files behind and leave the host program still functional.
- **Boot**. A boot virus infects the boot sector of the system, executing every time the system is booted and before the operating system is loaded. It watches for other bootable media and infects them. These viruses are also known as memory viruses, because they do not appear in the file system. Figure 16.5 shows how a boot virus works. Boot viruses have also adapted to infect firmware, such as network card PXE and Extensible Firmware Interface (EFI) environments.
- **Macro**. Most viruses are written in a low-level language, such as assembly or C. Macro viruses are written in a high-level language, such as Visual Basic. These viruses are triggered when a program capable of executing the macro is run. For example, a macro virus could be contained in a spreadsheet file.
- **Rootkit**. Originally coined to describe back doors on UNIX systems meant to provide easy root access, the term has since expanded to viruses and malware that infiltrate the operating system itself. The result is complete system compromise; no aspect of the system can be deemed trusted. When malware infects the operating system, it can take over all of the system's functions, including those functions that would normally facilitate its own detection.

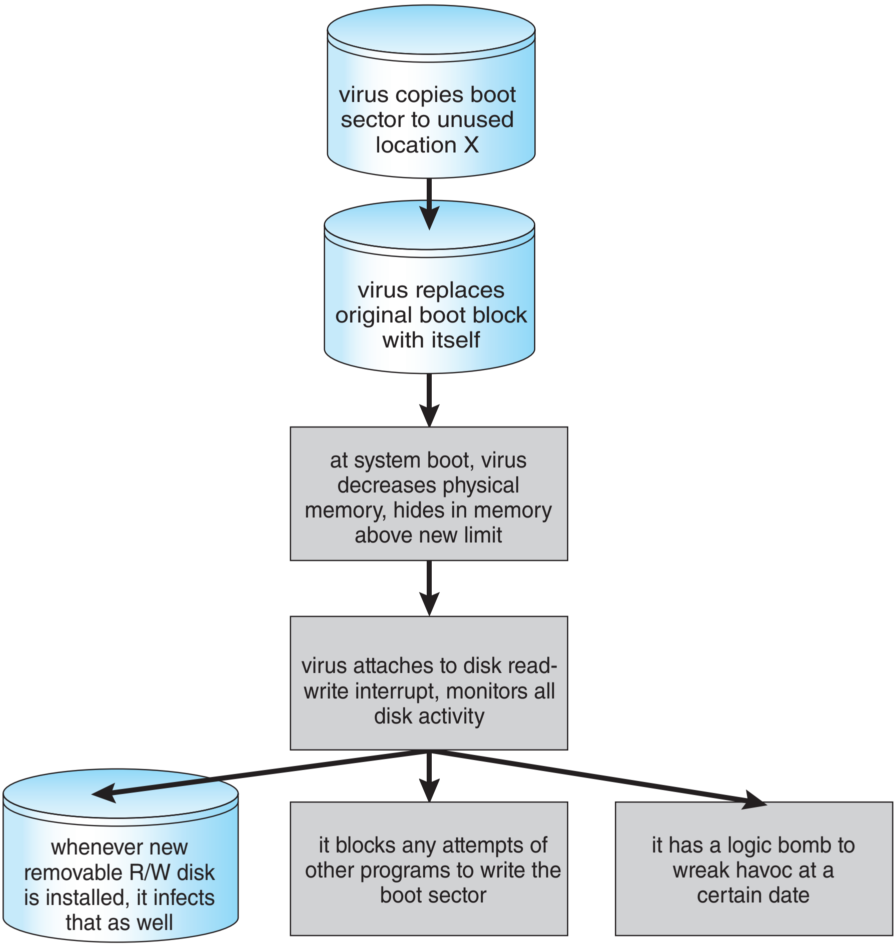

virus copies boot sector to unused location X
virus replaces original boot block with itself
at system boot, virus decreases physical memory, hides in memory above new limit
virus attaches to disk read-write interrupt, monitors all disk activity
whenever new removable R/W disk is installed, it infects that as well
it blocks any attempts of other programs to write the boot sector
it has a logic bomb to wreak havoc at a certain date

**Figure 16.5** A boot-sector computer virus.

- **Source code**. A source code virus looks for source code and modifies it to include the virus and to help spread the virus.
- **Polymorphic**. A polymorphic virus changes each time it is installed to avoid detection by antivirus software. The changes do not affect the virus's functionality but rather change the virus's signature. A **virus signature** is a pattern that can be used to identify a virus, typically a series of bytes that make up the virus code.
- **Encrypted**. An encrypted virus includes decryption code along with the encrypted virus, again to avoid detection. The virus first decrypts and then executes.
- **Stealth**. This tricky virus attempts to avoid detection by modifying parts of the system that could be used to detect it. For example, it could modify the `read` system call so that if the file it has modified is read, the original form of the code is returned rather than the infected code.

- **Multipartite**. A virus of this type is able to infect multiple parts of a system, including boot sectors, memory, and files. This makes it difficult to detect and contain.
- **Armored**. An armored virus is obfuscated—that is, written so as to be hard for antivirus researchers to unravel and understand. It can also be compressed to avoid detection and disinfection. In addition, virus droppers and other full files that are part of a virus infestation are frequently hidden via file attributes or unviewable file names.

This vast variety of viruses has continued to grow. For example, in 2004 a widespread virus was detected. It exploited three separate bugs for its operation. This virus started by infecting hundreds of Windows servers (including many trusted sites) running Microsoft Internet Information Server (IIS). Any vulnerable Microsoft Explorer web browser visiting those sites received a browser virus with any download. The browser virus installed several back-door programs, including a **keystroke logger**, which records everything entered on the keyboard (including passwords and credit-card numbers). It also installed a daemon to allow unlimited remote access by an intruder and another that allowed an intruder to route spam through the infected desktop computer.

An active security-related debate within the computing community concerns the existence of a **monoculture**, in which many systems run the same hardware, operating system, and application software. This monoculture supposedly consists of Microsoft products. One question is whether such a monoculture even exists today. Another question is whether, if it does, it increases the threat of and damage caused by viruses and other security intrusions. Vulnerability information is bought and sold in places like the **dark web** (World Wide Web systems reachable via unusual client configurations or methods). The more systems an attack can affect, the more valuable the attack.

## 16.3 System and Network Threats

Program threats, by themselves, pose serious security risks. But those risks are compounded by orders of magnitude when a system is connected to a network. Worldwide connectivity makes the system vulnerable to worldwide attacks.

The more ***open*** an operating system is—the more services it has enabled and the more functions it allows—the more likely it is that a bug is available to exploit it. Increasingly, operating systems strive to be **secure by default**. For example, Solaris 10 moved from a model in which many services (FTP, telnet, and others) were enabled by default when the system was installed to a model in which almost all services are disabled at installation time and must specifically be enabled by system administrators. Such changes reduce the system's attack surface.

All hackers leave tracks behind them—whether via network traffic patterns, unusual packet types, or other means. For that reason, hackers frequently launch attacks from **zombie systems**—independent systems or devices that have been compromised by hackers but that continue to serve their owners while being used without the owners' knowledge for nefarious purposes,

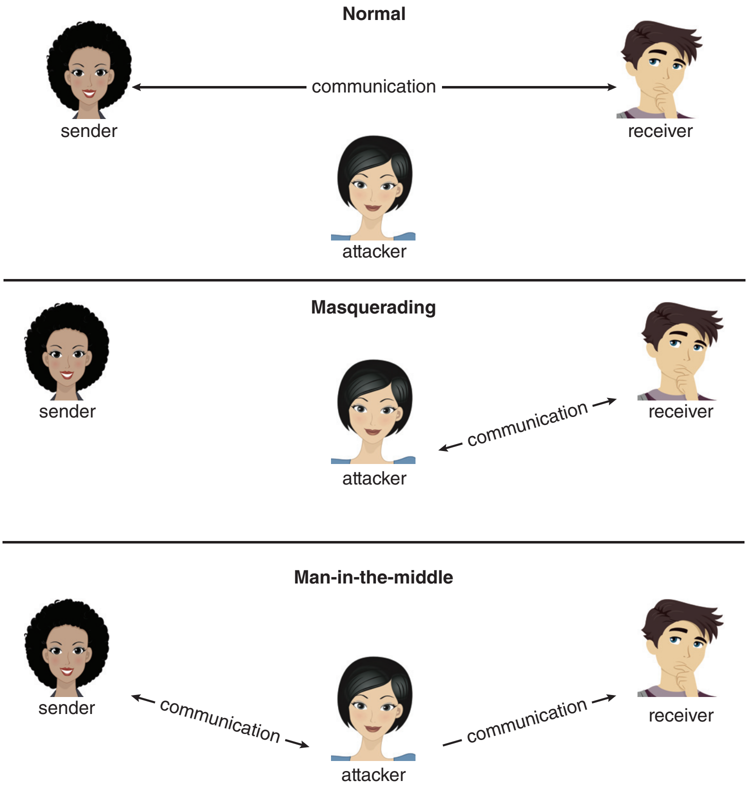

Normal
sender
communication
receiver
attacker
Masquerading
sender
attacker
communication
receiver
Man-in-the-middle
sender
communication
attacker
communication
receiver

**Figure 16.6** Standard security attacks.[1]

including denial-of-service attacks and spam relay. Zombies make hackers particularly difficult to track because they mask the original source of the attack and the identity of the attacker. This is one of many reasons for securing "inconsequential" systems, not just systems containing "valuable" information or services—lest they be turned into strongholds for hackers.

The widespread use of broadband and WiFi has only exacerbated the difficulty in tracking down attackers: even a simple desktop machine, which can often be easily compromised by malware, can become a valuable machine if used for its bandwidth or network access. Wireless ethernet makes it easy for attackers to launch attacks by joining a public network anonymously or "WarDriving"—locating a private unprotected network to target.

### 16.3.1 Attacking Network Traffic

Networks are common and attractive targets, and hackers have many options for mounting network attacks. As shown in Figure 16.6, an attacker can opt to remain passive and intercept network traffic (an attack commonly referred to as **sniffing**), often obtaining useful information about the types of sessions conducted between systems or the sessions' content. Alternatively, an attacker can take a more active role, either masquerading as one of the parties (referred to as **spoofing**), or becoming a fully active man-in-the-middle, intercepting and possibly modifying transactions between two peers.

---

[1] Lorelyn Medina/Shutterstock.

Next, we describe a common type of network attack, the denial-of-service (DoS) attack. Note that it is possible to guard against attacks through such means as encryption and authentication, which are discussed later in the chapter. Internet protocols do not, however, support either encryption or authentication by default.

### 16.3.2 Denial of Service

As mentioned earlier, denial-of-service attacks are aimed not at gaining information or stealing resources but rather at disrupting legitimate use of a system or facility. Most such attacks involve target systems or facilities that the attacker has not penetrated. Launching an attack that prevents legitimate use is frequently easier than breaking into a system or facility.

Denial-of-service attacks are generally network based. They fall into two categories. Attacks in the first category use so many facility resources that, in essence, no useful work can be done. For example, a website click could download a Java applet that proceeds to use all available CPU time or to pop up windows infinitely. The second category involves disrupting the network of the facility. There have been several successful denial-of-service attacks of this kind against major websites. Such attacks, which can last hours or days, have caused partial or full failure of attempts to use the target facility. The attacks are usually stopped at the network level until the operating systems can be updated to reduce their vulnerability.

Generally, it is impossible to prevent denial-of-service attacks. The attacks use the same mechanisms as normal operation. Even more difficult to prevent and resolve are **Distributed Denial-of-Service** (**DDoS**) attacks. These attacks are launched from multiple sites at once, toward a common target, typically by zombies. DDoS attacks have become more common and are sometimes associated with blackmail attempts. A site comes under attack, and the attackers offer to halt the attack in exchange for money.

Sometimes a site does not even know it is under attack. It can be difficult to determine whether a system slowdown is an attack or just a surge in system use. Consider that a successful advertising campaign that greatly increases traffic to a site could be considered a DDoS.

There are other interesting aspects of DoS attacks. For example, if an authentication algorithm locks an account for a period of time after several incorrect attempts to access the account, then an attacker could cause all authentication to be blocked by purposely making incorrect attempts to access all accounts. Similarly, a firewall that automatically blocks certain kinds of traffic could be induced to block that traffic when it should not. These examples suggest that programmers and systems managers need to fully understand the algorithms and technologies they are deploying. Finally, computer science classes are notorious sources of accidental system DoS attacks. Consider the first programming exercises in which students learn to create subprocesses or threads. A common bug involves spawning subprocesses infinitely. The system's free memory and CPU resources don't stand a chance.

### 16.3.3 Port Scanning

Port scanning is not itself an attack but is a means for a hacker to detect a system's vulnerabilities to attack. (Security personnel also use port scanning —for example, to detect services that are not needed or are not supposed to be running.) Port scanning typically is automated, involving a tool that attempts to create a TCP/IPconnection or send a UDP packet to a specific port or a range of ports.

Port scanning is often part of a reconnaissance technique known as fingerprinting, in which an attacker attempts to deduce the type of operating system in use and its set of services in order to identify known vulnerabilities. Many servers and clients make this easier by disclosing their exact version number as part of network protocol headers (for example, HTTP's "Server:" and "User-Agent:" headers). Detailed analyses of idiosyncratic behaviors by protocol handlers can also help the attacker figure out what operating system the target is using—a necessary step for successful exploitation.

Network vulnerability scanners are sold as commercial products. There are also tools that perform subsets of the functionality of a full scanner. For example, `nmap` (from http://www.insecure.org/nmap/) is a very versatile open-source utility for network exploration and security auditing. When pointed at a target, it will determine what services are running, including application names and versions. It can identify the host operating system. It can also provide information about defenses, such as what firewalls are defending the target. It does not exploit known bugs. Other tools, however (such as Metasploit), pick up where the port scanners leave off and provide payload construction facilities that can be used to test for vulnerabilities—or exploit them by creating a specific payload that triggers the bug.

The seminal work on port-scanning techniques can be found in http://phrack.org/issues/49/15.html. Techniques are constantly evolving, as are measures to detect them (which form the basis for network intrusion detection systems, discussed later).

## 16.4 Cryptography as a Security Tool

There are many defenses against computer attacks, running the gamut from methodology to technology. The broadest tool available to system designers and users is cryptography. In this section, we discuss cryptography and its use in computer security. Note that the cryptography discussed here has been simplified for educational purposes; readers are cautioned against using any of the schemes described here in the real world. Good cryptography libraries are widely available and would make a good basis for production applications.

In an isolated computer, the operating system can reliably determine the sender and recipient of all interprocess communication, since it controls all communication channels in the computer. In a network of computers, the situation is quite different. A networked computer receives bits "from the wire" with no immediate and reliable way of determining what machine or application sent those bits. Similarly, the computer sends bits onto the network with no way of knowing who might eventually receive them. Additionally, when either sending or receiving, the system has no way of knowing if an eavesdropper listened to the communication.

Commonly, network addresses are used to infer the potential senders and receivers of network messages. Network packets arrive with a source address, such as an IP address. And when a computer sends a message, it names the intended receiver by specifying a destination address. However, for applications where security matters, we are asking for trouble if we assume that the source or destination address of a packet reliably determines who sent or received that packet. A rogue computer can send a message with a falsified source address, and numerous computers other than the one specified by the destination address can (and typically do) receive a packet. For example, all of the routers on the way to the destination will receive the packet, too. How, then, is an operating system to decide whether to grant a request when it cannot trust the named source of the request? And how is it supposed to provide protection for a request or data when it cannot determine who will receive the response or message contents it sends over the network?

It is generally considered infeasible to build a network of any scale in which the source and destination addresses of packets can be ***trusted*** in this sense. Therefore, the only alternative is somehow to eliminate the need to trust the network. This is the job of cryptography. Abstractly, **cryptography** is used to constrain the potential senders and/or receivers of a message. Modern cryptography is based on secrets called **keys** that are selectively distributed to computers in a network and used to process messages. Cryptography enables a recipient of a message to verify that the message was created by some computer possessing a certain key. Similarly, a sender can encode its message so that only a computer with a certain key can decode the message. Unlike network addresses, however, keys are designed so that it is not computationally feasible to derive them from the messages they were used to generate or from any other public information. Thus, they provide a much more trustworthy means of constraining senders and receivers of messages.

Cryptography is a powerful tool, and the use of cryptography can cause contention. Some countries ban its use in certain forms or limit how long the keys can be. Others have ongoing debates about whether technology vendors (such as smartphone vendors) must provide a **back door** to the included cryptography, allowing law enforcement to bypass the privacy it provides. Many observers argue, however, that back doors are an intentional security weakness that could be exploited by attackers or even misused by governments.

Finally, note that cryptography is a field of study unto itself, with large and small complexities and subtleties. Here, we explore the most important aspects of the parts of cryptography that pertain to operating systems.

### 16.4.1 Encryption

Because it solves a wide variety of communication security problems, **encryption** is used frequently in many aspects of modern computing. It is used to send messages securely across a network, as well as to protect database data, files, and even entire disks from having their contents read by unauthorized entities. An encryption algorithm enables the sender of a message to ensure that only a computer possessing a certain key can read the message or to ensure that the writer of data is the only reader of the data. Encryption of messages is an ancient practice, of course, and there have been many encryption algorithms, dating back to ancient times. In this section, we describe important modern encryption principles and algorithms.

An encryption algorithm consists of the following components:

- A set $K$ of keys.
- A set $M$ of messages.
- A set $C$ of ciphertexts.
- An encrypting function $E : K \rightarrow (M \rightarrow C)$. That is, for each $k \in K$, $E_k$ is a function for generating ciphertexts from messages. Both $E$ and $E_k$ for any $k$ should be efficiently computable functions. Generally, $E_k$ is a randomized mapping from messages to ciphertexts.
- A decrypting function $D : K \rightarrow (C \rightarrow M)$. That is, for each $k \in K$, $D_k$ is a function for generating messages from ciphertexts. Both $D$ and $D_k$ for any $k$ should be efficiently computable functions.

An encryption algorithm must provide this essential property: given a ciphertext $c \in C$, a computer can compute $m$ such that $E_k(m) = c$ only if it possesses $k$. Thus, a computer holding $k$ can decrypt ciphertexts to the plaintexts used to produce them, but a computer not holding $k$ cannot decrypt ciphertexts. Since ciphertexts are generally exposed (for example, sent on a network), it is important that it be infeasible to derive $k$ from the ciphertexts.

There are two main types of encryption algorithms: symmetric and asymmetric. We discuss both types in the following sections.

#### 16.4.1.1 Symmetric Encryption

In a **symmetric encryption algorithm**, the same key is used to encrypt and to decrypt. Therefore, the secrecy of $k$ must be protected. Figure 16.7 shows an example of two users communicating securely via symmetric encryption over an insecure channel. Note that the key exchange can take place directly between the two parties or via a trusted third party (that is, a certificate authority), as discussed in Section 16.4.1.4.

For the past several decades, the most commonly used symmetric encryption algorithm in the United States for civilian applications has been the **data-encryption standard** (**DES**) cipher adopted by the National Institute of Standards and Technology (NIST). DES works by taking a 64-bit value and a 56-bit key and performing a series of transformations that are based on substitution and permutation operations. Because DES works on a block of bits at a time, is known as a **block cipher**, and its transformations are typical of block ciphers. With block ciphers, if the same key is used for encrypting an extended amount of data, it becomes vulnerable to attack.

DES is now considered insecure for many applications because its keys can be exhaustively searched with moderate computing resources. (Note, though, that it is still frequently used.) Rather than giving up on DES, NIST created a modification called **triple DES**, in which the DES algorithm is repeated three times (two encryptions and one decryption) on the same plaintext using two or three keys—for example, $c = E_{k3}(D_{k2}(E_{k1}(m)))$. When three keys are used, the effective key length is 168 bits.

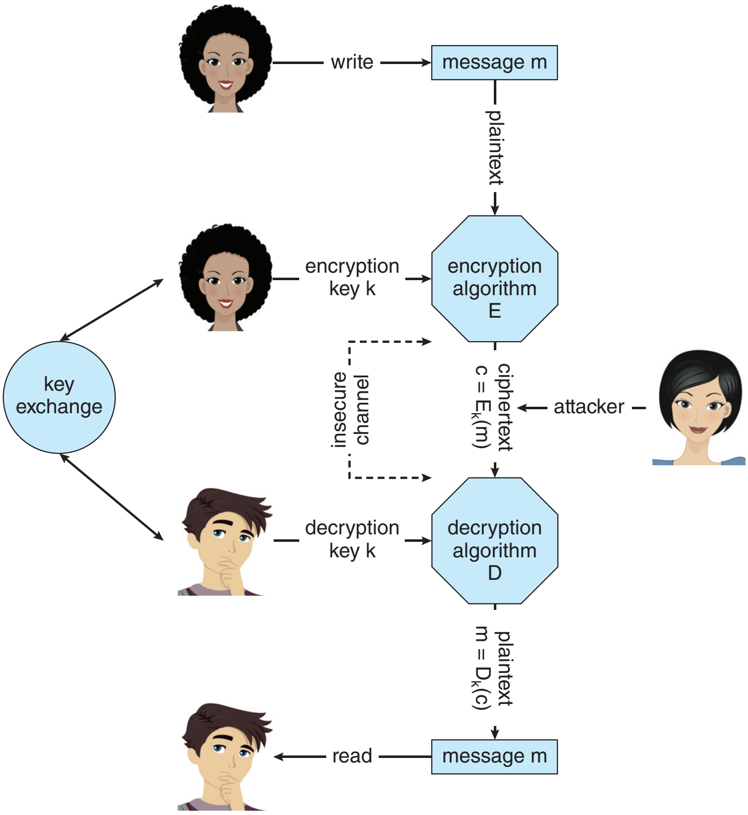

write
message m
plaintext
encryption key k
encryption algorithm E
key exchange
insecure channel
ciphertext c = E_k(m)
attacker
decryption key k
decryption algorithm D
plaintext m = D_k(c)
read
message m

**Figure 16.7** A secure communication over an insecure medium.[2]

In 2001, NIST adopted a new block cipher, called the **advanced encryption standard** (**AES**), to replace DES. AES (also known as Rijndael) has been standardized in FIPS-197 (http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf). It can use key lengths of 128, 192, or 256 bits and works on 128-bit blocks. Generally, the algorithm is compact and efficient.

Block ciphers are not necessarily secure encryption schemes. In particular, they do not directly handle messages longer than their required block sizes. An alternative is stream ciphers, which can be used to securely encrypt longer messages.

A **stream cipher** is designed to encrypt and decrypt a stream of bytes or bits rather than a block. This is useful when the length of a communication would make a block cipher too slow. The key is input into a pseudo−random-bit generator, which is an algorithm that attempts to produce random bits. The output of the generator when fed a key is a keystream. A **keystream** is an infinite set of bits that can be used to encrypt a plaintext stream through an XOR operation. (XOR, for “exclusive OR” is an operation that compares two input bits and generates one output bit. If the bits are the same, the result is 0. If the bits are different, the result is 1.) AES-based cipher suites include stream ciphers and are the most common today.

---

[2] Lorelyn Medina/Shutterstock.

#### 16.4.1.2 Asymmetric Encryption

In an **asymmetric encryption algorithm**, there are different encryption and decryption keys. An entity preparing to receive encrypted communication creates two keys and makes one of them (called the public key) available to anyone who wants it. Any sender can use that key to encrypt a communication, but only the key creator can decrypt the communication. This scheme, known as **public-key encryption**, was a breakthrough in cryptography (first described by Diffie and Hellman in https://www-ee.stanford.edu/ hellman/publications/24.pdf). No longer must a key be kept secret and delivered securely. Instead, anyone can encrypt a message to the receiving entity, and no matter who else is listening, only that entity can decrypt the message.

As an example of how public-key encryption works, we describe an algorithm known as **RSA**, after its inventors, Rivest, Shamir, and Adleman. RSA is the most widely used asymmetric encryption algorithm. (Asymmetric algorithms based on elliptic curves are gaining ground, however, because the key length of such an algorithm can be shorter for the same amount of cryptographic strength.)

In RSA, $k_e$ is the **public key**, and $k_d$ is the **private key**. $N$ is the product of two large, randomly chosen prime numbers $p$ and $q$ (for example, $p$ and $q$ are 2048 bits each). It must be computationally infeasible to derive $k_{d,N}$ from $k_{e,N}$, so that $k_e$ need not be kept secret and can be widely disseminated. The encryption algorithm is $E_{ke,N}(m) = m^{k_e} \bmod N$, where $k_e$ satisfies $k_e k_d \bmod (p-1)(q-1) = 1$. The decryption algorithm is then $D_{k_d,N}(c) = c^{k_d} \bmod N$.

An example using small values is shown in Figure 16.8. In this example, we make $p = 7$ and $q = 13$. We then calculate $N = 7 * 13 = 91$ and $(p-1)(q-1) = 72$. We next select $k_e$ relatively prime to 72 and $< 72$, yielding 5. Finally, we calculate $k_d$ such that $k_e k_d \bmod 72 = 1$, yielding 29. We now have our keys: the public key, $k_{e,N} = 5, 91$, and the private key, $k_{d,N} = 29, 91$. Encrypting the message 69 with the public key results in the message 62, which is then decoded by the receiver via the private key.

The use of asymmetric encryption begins with the publication of the public key of the destination. For bidirectional communication, the source also must publish its public key. “Publication” can be as simple as handing over an electronic copy of the key, or it can be more complex. The private key (or “secret key”) must be zealously guarded, as anyone holding that key can decrypt any message created by the matching public key.

We should note that the seemingly small difference in key use between asymmetric and symmetric cryptography is quite large in practice. Asymmetric cryptography is much more computationally expensive to execute. It is much faster for a computer to encode and decode ciphertext by using the usual symmetric algorithms than by using asymmetric algorithms. Why, then, use an asymmetric algorithm? In truth, these algorithms are not used for general-purpose encryption of large amounts of data. However, they are used not only for encryption of small amounts of data but also for authentication, confidentiality, and key distribution, as we show in the following sections.

#### 16.4.1.3 Authentication

We have seen that encryption offers a way of constraining the set of possible receivers of a message. Constraining the set of potential senders of a message

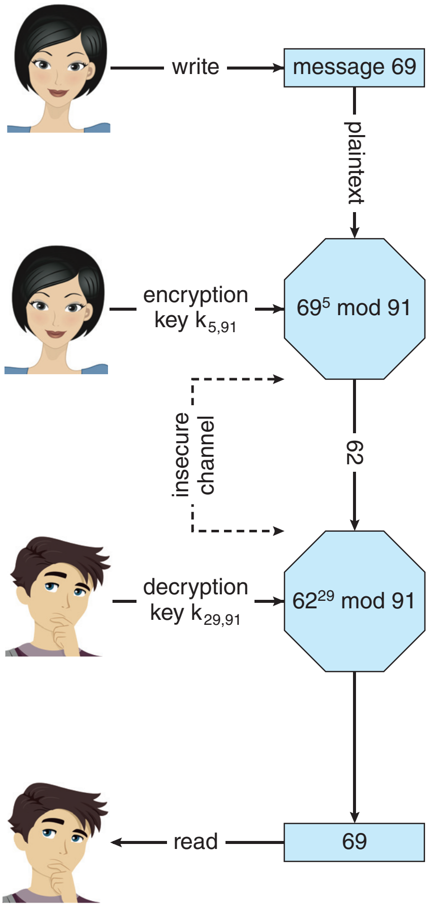

write
message 69
plaintext
encryption
key $k_{5,91}$
$69^5$ mod 91
insecure
channel
62
decryption
key $k_{29,91}$
$62^{29}$ mod 91
read
69

**Figure 16.8** Encryption and decryption using RSA asymmetric cryptography.[3]

is called **authentication**. Authentication is thus complementary to encryption. Authentication is also useful for proving that a message has not been modified. Next, we discuss authentication as a constraint on possible senders of a message. Note that this sort of authentication is similar to but distinct from user authentication, which we discuss in Section 16.5.

An authentication algorithm using symmetric keys consists of the following components:

- A set $K$ of keys.
- A set $M$ of messages.
- A set $A$ of authenticators.
- A function $S : K \rightarrow (M \rightarrow A)$. That is, for each $k \in K$, $S_k$ is a function for generating authenticators from messages. Both $S$ and $S_k$ for any $k$ should be efficiently computable functions.

---

[3] Lorelyn Medina/Shutterstock.

- A function $V : K \rightarrow (M \times A \rightarrow \{\texttt{true}, \texttt{false}\})$. That is, for each $k \in K$, $V_k$ is a function for verifying authenticators on messages. Both $V$ and $V_k$ for any $k$ should be efficiently computable functions.

The critical property that an authentication algorithm must possess is this: for a message $m$, a computer can generate an authenticator $a \in A$ such that $V_k(m, a) = \texttt{true}$ only if it possesses $k$. Thus, a computer holding $k$ can generate authenticators on messages so that any computer possessing $k$ can verify them. However, a computer not holding $k$ cannot generate authenticators on messages that can be verified using $V_k$. Since authenticators are generally exposed (for example, sent on a network with the messages themselves), it must not be feasible to derive $k$ from the authenticators. Practically, if $V_k(m, a) = \texttt{true}$, then we know that $m$ has not been modified and that the sender of the message has $k$. If we share $k$ with only one entity, then we know that the message originated from $k$.

Just as there are two types of encryption algorithms, there are two main varieties of authentication algorithms. The first step in understanding these algorithms is to explore hash functions. A **hash function** $H(m)$ creates a small, fixed-sized block of data, known as a **message digest** or **hash value**, from a message $m$. Hash functions work by taking a message, splitting it into blocks, and processing the blocks to produce an $n$-bit hash. $H$ must be collision resistant—that is, it must be infeasible to find an $m' \neq m$ such that $H(m) = H(m')$. Now, if $H(m) = H(m')$, we know that $m = m'$—that is, we know that the message has not been modified. Common message-digest functions include **MD5** (now considered insecure), which produces a 128-bit hash, and **SHA-1**, which outputs a 160-bit hash. Message digests are useful for detecting changed messages but are not useful as authenticators. For example, $H(m)$ can be sent along with a message; but if $H$ is known, then someone could modify $m$ to $m'$ and recompute $H(m')$, and the message modification would not be detected. Therefore, we must authenticate $H(m)$.

The first main type of authentication algorithm uses symmetric encryption. In a **message-authentication code** (**MAC**), a cryptographic checksum is generated from the message using a secret key. A MAC provides a way to securely authenticate short values. If we use it to authenticate $H(m)$ for an $H$ that is collision resistant, then we obtain a way to securely authenticate long messages by hashing them first. Note that $k$ is needed to compute both $S_k$ and $V_k$, so anyone able to compute one can compute the other.

The second main type of authentication algorithm is a **digital-signature algorithm**, and the authenticators thus produced are called **digital signatures**. Digital signatures are very useful in that they enable ***anyone*** to verify the authenticity of the message. In a digital-signature algorithm, it is computationally infeasible to derive $k_s$ from $k_v$. Thus, $k_v$ is the public key, and $k_s$ is the private key.

Consider as an example the RSA digital-signature algorithm. It is similar to the RSA encryption algorithm, but the key use is reversed. The digital signature of a message is derived by computing $S_{ks}(m) = H(m)^{k_s} \bmod N$. The key $k_s$ again is a pair $\langle d, N \rangle$, where $N$ is the product of two large, randomly chosen prime numbers $p$ and $q$. The verification algorithm is then $V_{kv}(m, a) \overset{?}{=} a^{k_v} \bmod N = H(m))$, where $k_v$ satisfies $k_v k_s \bmod (p-1)(q-1) = 1$. Digital signatures (as is the case with many aspects of cryptography) can be used on other entities than messages. For example creators of programs can “sign their code” via a digital signature to validate that the code has not been modified between its publication and its installation on a computer. **Code signing** has become a very common security improvement method on many systems.

Note that encryption and authentication may be used together or separately. Sometimes, for instance, we want authentication but not confidentiality. For example, a company could provide a software patch and could “sign” that patch to prove that it came from the company and that it hasn’t been modified.

Authentication is a component of many aspects of security. For example, digital signatures are the core of **nonrepudiation**, which supplies proof that an entity performed an action. A typical example of nonrepudiation involves the filling out of electronic forms as an alternative to the signing of paper contracts. Nonrepudiation assures that a person filling out an electronic form cannot deny that he did so.

#### 16.4.1.4 Key Distribution

Certainly, a good part of the battle between cryptographers (those inventing ciphers) and cryptanalysts (those trying to break them) involves keys. With symmetric algorithms, both parties need the key, and no one else should have it. The delivery of the symmetric key is a huge challenge. Sometimes it is performed **out-of-band**. For example, if Walter wanted to communicate with Rebecca securely, they could exchange a key via a paper document or a conversation and then have the communication electronically. These methods do not scale well, however. Also consider the key-management challenge. Suppose Lucy wanted to communicate with $N$ other users privately. Lucy would need $N$ keys and, for more security, would need to change those keys frequently.

These are the very reasons for efforts to create asymmetric key algorithms. Not only can the keys be exchanged in public, but a given user, say Audra, needs only one private key, no matter how many other people she wants to communicate with. There is still the matter of managing a public key for each recipient of the communication, but since public keys need not be secured, simple storage can be used for that **key ring**.

Unfortunately, even the distribution of public keys requires some care. Consider the man-in-the-middle attack shown in Figure 16.9. Here, the person who wants to receive an encrypted message sends out his public key, but an attacker also sends her “bad” public key (which matches her private key). The person who wants to send the encrypted message knows no better and so uses the bad key to encrypt the message. The attacker then happily decrypts it.

The problem is one of authentication—what we need is proof of who (or what) owns a public key. One way to solve that problem involves the use of digital certificates. A **digital certificate** is a public key digitally signed by a trusted party. The trusted party receives proof of identification from some entity and certifies that the public key belongs to that entity. But how do we know we can trust the certifier? These **certificate authorities** have their public keys included within web browsers (and other consumers of certificates) before they are distributed. The certificate authorities can then vouch for other authorities (digitally signing the public keys of these other authorities), and so on, creating a web of trust. The certificates can be distributed in a standard

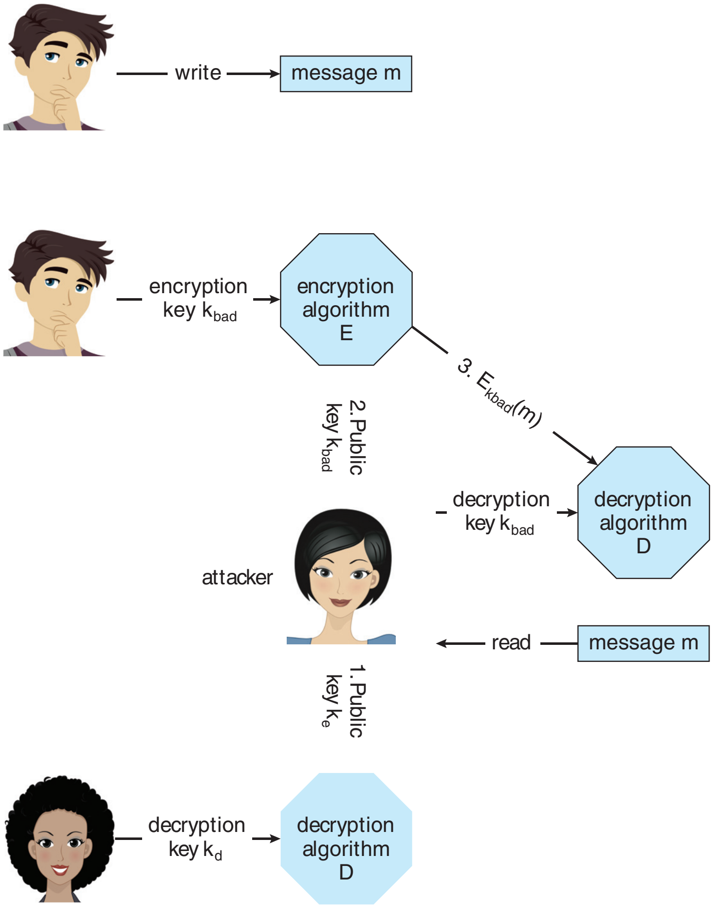

write
message m
encryption key $k_{bad}$
encryption algorithm E
3. $E_{k_{bad}}(m)$
2. Public key $k_{bad}$
decryption key $k_{bad}$
decryption algorithm D
attacker
read
message m
1. Public key $k_e$
decryption key $k_d$
decryption algorithm D

**Figure 16.9** A man-in-the-middle attack on asymmetric cryptography.[4]

X.509 digital certificate format that can be parsed by computer. This scheme is used for secure web communication, as we discuss in Section 16.4.3.

### 16.4.2 Implementation of Cryptography

Network protocols are typically organized in ***layers***, with each layer acting as a client of the one below it. That is, when one protocol generates a message to send to its protocol peer on another machine, it hands its message to the protocol below it in the network-protocol stack for delivery to its peer on that machine. For example, in an IP network, TCP (a ***transport-layer*** protocol) acts as a client of IP (a ***network-layer*** protocol): TCP packets are passed down to IP for delivery to the IP peer at the other end of the connection. IP encapsulates the TCP packet in an IP packet, which it similarly passes down to the ***data-link layer*** to be transmitted across the network to its peer on the destination computer. This IP peer then delivers the TCP packet up to the TCP peer on that machine. Seven such layers are included in the OSI model, mentioned earlier and described in detail in Section 19.3.2.

---

[4] Lorelyn Medina/Shutterstock.

Cryptography can be inserted at almost any layer in network protocol stacks. TLS (Section 16.4.3), for example, provides security at the transport layer. Network-layer security generally has been standardized on **IPSec**, which defines IP packet formats that allow the insertion of authenticators and the encryption of packet contents. IPSec uses symmetric encryption and uses the **Internet Key Exchange** (**IKE**) protocol for key exchange. IKE is based on public-key encryption. IPSec has widely used as the basis for **virtual private networks** (**VPNs**), in which all traffic between two IPSec endpoints is encrypted to make a private network out of one that would otherwise be public. Numerous protocols also have been developed for use by applications, such as PGP for encrypting e-mail; in this type of scheme, the applications themselves must be coded to implement security.

Where is cryptographic protection best placed in a protocol stack? In general, there is no definitive answer. On the one hand, more protocols benefit from protections placed lower in the stack. For example, since IP packets encapsulate TCP packets, encryption of IP packets (using IPSec, for example) also hides the contents of the encapsulated TCP packets. Similarly, authenticators on IP packets detect the modification of contained TCP header information.

On the other hand, protection at lower layers in the protocol stack may give insufficient protection to higher-layer protocols. For example, an application server that accepts connections encrypted with IPSec might be able to authenticate the client computers from which requests are received. However, to authenticate a user at a client computer, the server may need to use an application-level protocol—the user may be required to type a password. Also consider the problem of e-mail. E-mail delivered via the industry-standard SMTP protocol is stored and forwarded, frequently multiple times, before it is delivered. Each of these transmissions could go over a secure or an insecure network. For e-mail to be secure, the e-mail message needs to be encrypted so that its security is independent of the transports that carry it.

Unfortunately, like many tools, encryption can be used not only for "good" but also for "evil." The ransomware attacks described earlier, for example, are based on encryption. As mentioned, the attackers encrypt information on the target system and render it inaccessible to the owner. The idea is to force the owner to pay a ransom to get the key needed to decrypt the data. Prevention of such attacks takes the form of better system and network security and a well-executed backup plan so that the contents of the files can be restored without the key.

### 16.4.3 An Example: TLS

**Transport Layer Security** (**TLS**) is a cryptographic protocol that enables two computers to communicate securely—that is, so that each can limit the sender and receiver of messages to the other. It is perhaps the most commonly used cryptographic protocol on the Internet today, since it is the standard protocol by which web browsers communicate securely with web servers. For completeness, we should note that TLS evolved from SSL (Secure Sockets Layer), which was designed by Netscape. It is described in detail in https://tools.ietf.org/html/rfc5246.

TLS is a complex protocol with many options. Here, we present only a single variation of it. Even then, we describe it in a very simplified and abstract form, so as to maintain focus on its use of cryptographic primitives. What we are about to see is a complex dance in which asymmetric cryptography is used so that a client and a server can establish a secure **session key** that can be used for symmetric encryption of the session between the two—all of this while avoiding man-in-the-middle and replay attacks. For added cryptographic strength, the session keys are forgotten once a session is completed. Another communication between the two will require generation of new session keys.

The TLS protocol is initiated by a client $c$ to communicate securely with a server. Prior to the protocol's use, the server $s$ is assumed to have obtained a certificate, denoted $\mathsf{cert}_s$, from certification authority CA. This certificate is a structure containing the following:

- Various attributes (attrs) of the server, such as its unique ***distinguished*** name and its ***common*** (DNS) name
- The identity of a asymmetric encryption algorithm $E()$ for the server
- The public key $k_e$ of this server
- A validity interval (interval) during which the certificate should be considered valid
- A digital signature a on the above information made by the CA—that is, $\mathsf{a} = \mathsf{S}_{kCA}(\langle\ \mathsf{attrs}, \mathrm{E}_{ke}, \mathsf{interval}\ \rangle)$

In addition, prior to the protocol's use, the client is presumed to have obtained the public verification algorithm $V_{kCA}$ for CA. In the case of the web, the user's browser is shipped from its vendor containing the verification algorithms and public keys of certain certification authorities. The user can delete these or add others.

When $c$ connects to $s$, it sends a 28-byte random value $n_c$ to the server, which responds with a random value $n_s$ of its own, plus its certificate $\mathsf{cert}_s$. The client verifies that $V_{kCA}(\langle\ \mathsf{attrs}, E_{ke}, \mathsf{interval}\rangle, \mathrm{a}) = \mathtt{true}$ and that the current time is in the validity interval interval. If both of these tests are satisfied, the server has proved its identity. Then the client generates a random 46-byte ***premaster secret*** pms and sends $\mathsf{cpms} = E_{ke}(\mathsf{pms})$ to the server. The server recovers $\mathsf{pms} = D_{kd}(\mathsf{cpms})$. Now both the client and the server are in possession of $n_c$, $n_s$, and pms, and each can compute a shared 48-byte ***master secret*** $\mathsf{ms} = H(n_c, n_s, \mathsf{pms})$. Only the server and client can compute ms, since only they know pms. Moreover, the dependence of ms on $n_c$ and $n_s$ ensures that ms is a ***fresh*** value —that is, a session key that has not been used in a previous communication. At this point, the client and the server both compute the following keys from the ms:

- A symmetric encryption key $k_{cs}^{\mathsf{crypt}}$ for encrypting messages from the client to the server
- A symmetric encryption key $k_{sc}^{\mathsf{crypt}}$ for encrypting messages from the server to the client
- A MAC generation key $k_{cs}^{\mathsf{mac}}$ for generating authenticators on messages from the client to the server

- A MAC generation key $k_{sc}^{\mathsf{mac}}$ for generating authenticators on messages from the server to the client

To send a message $m$ to the server, the client sends

$$c = E_{k_{cs}^{\mathsf{crypt}}}(\langle m, S_{k_{cs}^{\mathsf{mac}}}(m)\rangle).$$

Upon receiving $c$, the server recovers

$$\langle m, a\rangle = D_{k_{cs}^{\mathsf{crypt}}}(c)$$

and accepts $m$ if $V_{k_{cs}^{\mathsf{mac}}}(m, a) = \text{true}$. Similarly, to send a message $m$ to the client, the server sends

$$c = E_{k_{sc}^{\mathsf{crypt}}}(\langle m, S_{k_{sc}^{\mathsf{mac}}}(m)\rangle)$$

and the client recovers

$$\langle m, a\rangle = D_{k_{sc}^{\mathsf{crypt}}}(c)$$

and accepts $m$ if $V_{k_{sc}^{\mathsf{mac}}}(m, a) = \text{true}$.

This protocol enables the server to limit the recipients of its messages to the client that generated pms and to limit the senders of the messages it accepts to that same client. Similarly, the client can limit the recipients of the messages it sends and the senders of the messages it accepts to the party that knows $k_d$ (that is, the party that can decrypt cpms). In many applications, such as web transactions, the client needs to verify the identity of the party that knows $k_d$. This is one purpose of the certificate $\mathsf{cert}_s$. In particular, the attrs field contains information that the client can use to determine the identity—for example, the domain name—of the server with which it is communicating. For applications in which the server also needs information about the client, TLS supports an option by which a client can send a certificate to the server.

In addition to its use on the Internet, TLS is being used for a wide variety of tasks. For example, we mentioned earlier that IPSec is widely used as the basis for virtual private networks, or VPNs. IPSec VPNs now have a competitor in TLS VPNs. IPSec is good for point-to-point encryption of traffic—say, between two company offices. TLS VPNs are more flexible but not as efficient, so they might be used between an individual employee working remotely and the corporate office.

## 16.5 User Authentication

Our earlier discussion of authentication involves messages and sessions. But what about users? If a system cannot authenticate a user, then authenticating that a message came from that user is pointless. Thus, a major security problem for operating systems is **user authentication**. The protection system depends on the ability to identify the programs and processes currently executing, which in turn depends on the ability to identify each user of the system. Users normally identify themselves, but how do we determine whether a user's identity is authentic? Generally, user authentication is based on one or more of three things: the user's possession of something (a key or card), the user's knowledge of something (a user identifier and password), or an attribute of the user (fingerprint, retina pattern, or signature).

### 16.5.1 Passwords

The most common approach to authenticating a user identity is the use of **passwords**. When the user identifies herself by user ID or account name, she is asked for a password. If the user-supplied password matches the password stored in the system, the system assumes that the account is being accessed by the owner of that account.

Passwords are often used to protect objects in the computer system, in the absence of more complete protection schemes. They can be considered a special case of either keys or capabilities. For instance, a password may be associated with each resource (such as a file). Whenever a request is made to use the resource, the password must be given. If the password is correct, access is granted. Different passwords may be associated with different access rights. For example, different passwords may be used for reading files, appending files, and updating files.

In practice, most systems require only one password for a user to gain their full rights. Although more passwords theoretically would be more secure, such systems tend not to be implemented due to the classic trade-off between security and convenience. If security makes something inconvenient, then the security is frequently bypassed or otherwise circumvented.

### 16.5.2 Password Vulnerabilities

Passwords are extremely common because they are easy to understand and use. Unfortunately, passwords can often be guessed, accidentally exposed, sniffed (read by an eavesdropper), or illegally transferred from an authorized user to an unauthorized one, as we show next.

There are three common ways to guess a password. One way is for the intruder (either human or program) to know the user or to have information about the user. All too frequently, people use obvious information (such as the names of their cats or spouses) as their passwords. Another way is to use brute force, trying enumeration—or all possible combinations of valid password characters (letters, numbers, and punctuation on some systems)—until the password is found. Short passwords are especially vulnerable to this method. For example, a four-character password provides only 10,000 variations. On average, guessing 5,000 times would produce a correct hit. A program that could try a password every millisecond would take only about 5 seconds to guess a four-character password. Enumeration is less successful where systems allow longer passwords that include both uppercase and lowercase letters, along with numbers and all punctuation characters. Of course, users must take advantage of the large password space and must not, for example, use only lowercase letters. The third, common method is dictionary attacks where all words, word variations, and common passwords are tried.

In addition to being guessed, passwords can be exposed as a result of visual or electronic monitoring. An intruder can look over the shoulder of a user (**shoulder surfing**) when the user is logging in and can learn the password easily by watching the keyboard. Alternatively, anyone with access to the network on which a computer resides can seamlessly add a network monitor, allowing him to **sniff**, or watch, all data being transferred on the network, including user IDs and passwords. Encrypting the data stream containing the password solves this problem. Even such a system could have passwords stolen, however. For example, if a file is used to contain the passwords, it could be copied for off-system analysis. Or consider a Trojan-horse program installed on the system that captures every keystroke before sending it on to the application. Another common method to grab passwords, specially debit card passcodes, is installing physical devices where the codes are used and recording what the user does, for example a “skimmer” at an ATM machine or a device installed between the keyboard and the computer.

Exposure is a particularly severe problem if the password is written down where it can be read or lost. Some systems force users to select hard-to-remember or long passwords, or to change their password frequently, which may cause a user to record the password or to reuse it. As a result, such systems provide much less security than systems that allow users to select easy passwords!

The final type of password compromise, illegal transfer, is the result of human nature. Most computer installations have a rule that forbids users to share accounts. This rule is sometimes implemented for accounting reasons but is often aimed at improving security. For instance, suppose one user ID is shared by several users, and a security breach occurs from that user ID. It is impossible to know who was using the ID at the time the break occurred or even whether the user was an authorized one. With one user per user ID, any user can be questioned directly about use of the account; in addition, the user might notice something different about the account and detect the break-in. Sometimes, users break account-sharing rules to help friends or to circumvent accounting, and this behavior can result in a system’s being accessed by unauthorized users—possibly harmful ones.

Passwords can be either generated by the system or selected by a user. System-generated passwords may be difficult to remember, and thus users may write them down. As mentioned, however, user-selected passwords are often easy to guess (the user’s name or favorite car, for example). Some systems will check a proposed password for ease of guessing or cracking before accepting it. Some systems also ***age*** passwords, forcing users to change their passwords at regular intervals (every three months, for instance). This method is not foolproof either, because users can easily toggle between two passwords. The solution, as implemented on some systems, is to record a password history for each user. For instance, the system could record the last $N$ passwords and not allow their reuse.

Several variants on these simple password schemes can be used. For example, the password can be changed more frequently. At the extreme, the password is changed from session to session. A new password is selected (either by the system or by the user) at the end of each session, and that password must be used for the next session. In such a case, even if a password is used by an unauthorized person, that person can use it only once. When the legitimate user tries to use a now-invalid password at the next session, he discovers the security violation. Steps can then be taken to repair the breached security.

### 16.5.3 Securing Passwords

One problem with all these approaches is the difficulty of keeping the password secret within the computer. How can the system store a password securely yet allow its use for authentication when the user presents her pass-

### *STRONG AND EASY TO REMEMBER PASSWORDS*

It is extremely important to use strong (hard to guess and hard to shoulder surf) passwords on critical systems like bank accounts. It is also important to not use the same password on lots of systems, as one less important, easily hacked system could reveal the password you use on more important systems. A good technique is to generate your password by using the first letter of each word of an easily remembered phrase using both upper and lower characters with a number or punctuation mark thrown in for good measure. For example, the phrase "My girlfriend's name is Katherine" might yield the password "Mgn.isK!". The password is hard to crack but easy for the user to remember. A more secure system would allow more characters in its passwords. Indeed, a system might also allow passwords to include the space character, so that a user could create a **passphrase** which is easy to remember but difficult to break.

word? The UNIX system uses secure hashing to avoid the necessity of keeping its password list secret. Because the password is hashed rather than encrypted, it is impossible for the system to decrypt the stored value and determine the original password.

Hash functions are easy to compute, but hard (if not impossible) to invert. That is, given a value $x$, it is easy to compute the hash function value $f(x)$. Given a function value $f(x)$, however, it is impossible to compute $x$. This function is used to encode all passwords. Only encoded passwords are stored. When a user presents a password, it is hashed and compared against the stored encoded password. Even if the stored encoded password is seen, it cannot be decoded, so the password cannot be determined. Thus, the password file does not need to be kept secret.

The drawback to this method is that the system no longer has control over the passwords. Although the passwords are hashed, anyone with a copy of the password file can run fast hash routines against it—hashing each word in a dictionary, for instance, and comparing the results against the passwords. If the user has selected a password that is also a word in the dictionary, the password is cracked. On sufficiently fast computers, or even on clusters of slow computers, such a comparison may take only a few hours. Furthermore, because systems use well-known hashing algorithms, an attacker might keep a cache of passwords that have been cracked previously.

For these reasons, systems include a "salt," or recorded random number, in the hashing algorithm. The salt value is added to the password to ensure that if two plaintext passwords are the same, they result in different hash values. In addition, the salt value makes hashing a dictionary ineffective, because each dictionary term would need to be combined with each salt value for comparison to the stored passwords. Newer versions of UNIX also store the hashed password entries in a file readable only by the superuser. The programs that compare the hash to the stored value run `setuid` to root, so they can read this file, but other users cannot.

### 16.5.4 One-Time Passwords

To avoid the problems of password sniffing and shoulder surfing, a system can use a set of **paired passwords**. When a session begins, the system randomly selects and presents one part of a password pair; the user must supply the other part. In this system, the user is ***challenged*** and must ***respond*** with the correct answer to that challenge.

This approach can be generalized to the use of an algorithm as a password. In this scheme, the system and the user share a symmetric password. The password $pw$ is never transmitted over a medium that allows exposure. Rather, the password is used as input to a function, along with a ***challenge*** $ch$ presented by the system. The user then computes the function $H(pw, ch)$. The result of this function is transmitted as the authenticator to the computer. Because the computer also knows $pw$ and $ch$, it can perform the same computation. If the results match, the user is authenticated. The next time the user needs to be authenticated, another $ch$ is generated, and the same steps ensue. This time, the authenticator is different. Such algorithmic passwords are not susceptible to reuse. That is, a user can type in a password, and no entity intercepting that password will be able to reuse it. This **one-time password** system is one of only a few ways to prevent improper authentication due to password exposure.

One-time password systems are implemented in various ways. Commercial implementations use hardware calculators with a display or a display and numeric keypad. These calculators generally take the shape of a credit card, a key-chain dongle, or a USB device. Software running on computers or smartphones provides the user with $H(pw, ch)$; $pw$ can be input by the user or generated by the calculator in synchronization with the computer. Sometimes, $pw$ is just a **personal identification number** (**PIN**). The output of any of these systems shows the one-time password. A one-time password generator that requires input by the user involves **two-factor authentication**. Two different types of components are needed in this case—for example, a one-time password generator that generates the correct response only if the PIN is valid. Two-factor authentication offers far better authentication protection than single-factor authentication because it requires "something you have" as well as "something you know."

### 16.5.5 Biometrics

Yet another variation on the use of passwords for authentication involves the use of biometric measures. Palm- or hand-readers are commonly used to secure physical access—for example, access to a data center. These readers match stored parameters against what is being read from hand-reader pads. The parameters can include a temperature map, as well as finger length, finger width, and line patterns. These devices are currently too large and expensive to be used for normal computer authentication.

Fingerprint readers have become accurate and cost-effective. These devices read finger ridge patterns and convert them into a sequence of numbers. Over time, they can store a set of sequences to adjust for the location of the finger on the reading pad and other factors. Software can then scan a finger on the pad and compare its features with these stored sequences to determine if they match. Of course, multiple users can have profiles stored, and the scanner can differentiate among them. A very accurate two-factor authentication scheme can result from requiring a password as well as a user name and fingerprint scan. If this information is encrypted in transit, the system can be very resistant to spoofing or replay attack.

**Multifactor authentication** is better still. Consider how strong authentication can be with a USB device that must be plugged into the system, a PIN, and a fingerprint scan. Except for having to place one's finger on a pad and plug the USB into the system, this authentication method is no less convenient than that using normal passwords. Recall, though, that strong authentication by itself is not sufficient to guarantee the ID of the user. An authenticated session can still be hijacked if it is not encrypted.

## 16.6 Implementing Security Defenses

Just as there are myriad threats to system and network security, there are many security solutions. The solutions range from improved user education, through technology, to writing better software. Most security professionals subscribe to the theory of **defense in depth**, which states that more layers of defense are better than fewer layers. Of course, this theory applies to any kind of security. Consider the security of a house without a door lock, with a door lock, and with a lock and an alarm. In this section, we look at the major methods, tools, and techniques that can be used to improve resistance to threats. Note that some security-improving techniques are more properly part of protection than security and are covered in Chapter 17.

### 16.6.1 Security Policy

The first step toward improving the security of any aspect of computing is to have a **security policy**. Policies vary widely but generally include a statement of what is being secured. For example, a policy might state that all outside-accessible applications must have a code review before being deployed, or that users should not share their passwords, or that all connection points between a company and the outside must have port scans run every six months. Without a policy in place, it is impossible for users and administrators to know what is permissible, what is required, and what is not allowed. The policy is a road map to security, and if a site is trying to move from less secure to more secure, it needs a map to know how to get there.

Once the security policy is in place, the people it affects should know it well. It should be their guide. The policy should also be a **living document** that is reviewed and updated periodically to ensure that it is still pertinent and still followed.

### 16.6.2 Vulnerability Assessment

How can we determine whether a security policy has been correctly implemented? The best way is to execute a vulnerability assessment. Such assessments can cover broad ground, from social engineering through risk assessment to port scans. **Risk assessment**, for example, attempts to value the assets of the entity in question (a program, a management team, a system, or a facility) and determine the odds that a security incident will affect the entity and decrease its value. When the odds of suffering a loss and the amount of the potential loss are known, a value can be placed on trying to secure the entity.

The core activity of most vulnerability assessments is a **penetration test**, in which the entity is scanned for known vulnerabilities. Because this book is concerned with operating systems and the software that runs on them, we concentrate on those aspects of vulnerability assessment.

Vulnerability scans typically are done at times when computer use is relatively low, to minimize their impact. When appropriate, they are done on test systems rather than production systems, because they can induce unhappy behavior from the target systems or network devices.

A scan within an individual system can check a variety of aspects of the system:

- Short or easy-to-guess passwords
- Unauthorized privileged programs, such as setuid programs
- Unauthorized programs in system directories
- Unexpectedly long-running processes
- Improper directory protections on user and system directories
- Improper protections on system data files, such as the password file, device files, or the operating-system kernel itself
- Dangerous entries in the program search path (for example, the Trojan horse discussed in Section 16.2.1), such as the current directory and any easily-written directories such as `/tmp`
- Changes to system programs detected with checksum values
- Unexpected or hidden network daemons

Any problems found by a security scan can be either fixed automatically or reported to the managers of the system.

Networked computers are much more susceptible to security attacks than are standalone systems. Rather than attacks from a known set of access points, such as directly connected terminals, we face attacks from an unknown and large set of access points—a potentially severe security problem. To a lesser extent, systems connected to telephone lines via modems are also more exposed.

In fact, the U.S. government considers a system to be only as secure as its most far-reaching connection. For instance, a top-secret system may be accessed only from within a building also considered top-secret. The system loses its top-secret rating if any form of communication can occur outside that environment. Some government facilities take extreme security precautions. The connectors that plug a terminal into the secure computer are locked in a safe in the office when the terminal is not in use. A person must have proper ID to gain access to the building and her office, must know a physical lock combination, and must know authentication information for the computer itself to gain access to the computer—an example of multifactor authentication.

Unfortunately for system administrators and computer-security professionals, it is frequently impossible to lock a machine in a room and disallow all remote access. For instance, the Internet currently connects billions of computers and devices and has become a mission-critical, indispensable resource for many companies and individuals. If you consider the Internet a club, then, as in any club with millions of members, there are many good members and some bad members. The bad members have many tools they can use to attempt to gain access to the interconnected computers.

Vulnerability scans can be applied to networks to address some of the problems with network security. The scans search a network for ports that respond to a request. If services are enabled that should not be, access to them can be blocked, or they can be disabled. The scans then determine the details of the application listening on that port and try to determine if it has any known vulnerabilities. Testing those vulnerabilities can determine if the system is misconfigured or lacks needed patches.

Finally, though, consider the use of port scanners in the hands of an attacker rather than someone trying to improve security. These tools could help attackers find vulnerabilities to attack. (Fortunately, it is possible to detect port scans through anomaly detection, as we discuss next.) It is a general challenge to security that the same tools can be used for good and for harm. In fact, some people advocate **security through obscurity**, stating that no tools should be written to test security, because such tools can be used to find (and exploit) security holes. Others believe that this approach to security is not a valid one, pointing out, for example, that attackers could write their own tools. It seems reasonable that security through obscurity be considered one of the layers of security only so long as it is not the only layer. For example, a company could publish its entire network configuration, but keeping that information secret makes it harder for intruders to know what to attack. Even here, though, a company assuming that such information will remain a secret has a false sense of security.

### 16.6.3 Intrusion Prevention

Securing systems and facilities is intimately linked to intrusion detection and prevention. **Intrusion prevention**, as its name suggests, strives to detect attempted or successful intrusions into computer systems and to initiate appropriate responses to the intrusions. Intrusion prevention encompasses a wide array of techniques that vary on a number of axes, including the following:

- The time at which detection occurs. Detection can occur in real time (while the intrusion is occurring) or after the fact.
- The types of inputs examined to detect intrusive activity. These may include user-shell commands, process system calls, and network packet headers or contents. Some forms of intrusion might be detected only by correlating information from several such sources.
- The range of response capabilities. Simple forms of response include alerting an administrator to the potential intrusion or somehow halting the potentially intrusive activity—for example, killing a process engaged in such activity. In a sophisticated form of response, a system might transparently divert an intruder's activity to a **honeypot**—a false resource exposed

to the attacker. The resource appears real to the attacker and enables the system to monitor and gain information about the attack.

These degrees of freedom in the design space for detecting intrusions have yielded a wide range of solutions, known as **intrusion-prevention systems** (**IPS**). IPSs act as self-modifying firewalls, passing traffic unless an intrusion is detected (at which point that traffic is blocked).

But just what constitutes an intrusion? Defining a suitable specification of intrusion turns out to be quite difficult, and thus automatic IPSs today typically settle for one of two less ambitious approaches. In the first, called **signature-based detection**, system input or network traffic is examined for specific behavior patterns (or **signatures**) known to indicate attacks. A simple example of signature-based detection is scanning network packets for the string "`/etc/passwd`" targeted for a UNIX system. Another example is virus-detection software, which scans binaries or network packets for known viruses.

The second approach, typically called **anomaly detection**, attempts through various techniques to detect anomalous behavior within computer systems. Of course, not all anomalous system activity indicates an intrusion, but the presumption is that intrusions often induce anomalous behavior. An example of anomaly detection is monitoring system calls of a daemon process to detect whether the system-call behavior deviates from normal patterns, possibly indicating that a buffer overflow has been exploited in the daemon to corrupt its behavior. Another example is monitoring shell commands to detect anomalous commands for a given user or detecting an anomalous login time for a user, either of which may indicate that an attacker has succeeded in gaining access to that user's account.

Signature-based detection and anomaly detection can be viewed as two sides of the same coin. Signature-based detection attempts to characterize dangerous behaviors and to detect when one of these behaviors occurs, whereas anomaly detection attempts to characterize normal (or nondangerous) behaviors and to detect when something other than these behaviors occurs.

These different approaches yield IPSs with very different properties, however. In particular, anomaly detection can find previously unknown methods of intrusion (so-called **zero-day attacks**). Signature-based detection, in contrast, will identify only known attacks that can be codified in a recognizable pattern. Thus, new attacks that were not contemplated when the signatures were generated will evade signature-based detection. This problem is well known to vendors of virus-detection software, who must release new signatures with great frequency as new viruses are detected manually.

Anomaly detection is not necessarily superior to signature-based detection, however. Indeed, a significant challenge for systems that attempt anomaly detection is to benchmark "normal" system behavior accurately. If the system has already been penetrated when it is benchmarked, then the intrusive activity may be included in the "normal" benchmark. Even if the system is benchmarked cleanly, without influence from intrusive behavior, the benchmark must give a fairly complete picture of normal behavior. Otherwise, the number of **false positives** (false alarms) or, worse, **false negatives** (missed intrusions) will be excessive.

To illustrate the impact of even a marginally high rate of false alarms, consider an installation consisting of a hundred UNIX workstations from which security-relevant events are recorded for purposes of intrusion detection. A small installation such as this could easily generate a million audit records per day. Only one or two might be worthy of an administrator's investigation. If we suppose, optimistically, that each actual attack is reflected in ten audit records, we can roughly compute the rate of occurrence of audit records reflecting truly intrusive activity as follows:

$$\frac{2\frac{\text{intrusions}}{\text{day}} \cdot 10\frac{\text{records}}{\text{intrusion}}}{10^6\frac{\text{records}}{\text{day}}} = 0.00002.$$

Interpreting this as a "probability of occurrence of intrusive records," we denote it as $P(I)$; that is, event $I$ is the occurrence of a record reflecting truly intrusive behavior. Since $P(I) = 0.00002$, we also know that $P(\neg I) = 1 - P(I) = 0.99998$. Now we let $A$ denote the raising of an alarm by an IDS. An accurate IDS should maximize both $P(I|A)$ and $P(\neg I|\neg A)$—that is, the probabilities that an alarm indicates an intrusion and that no alarm indicates no intrusion. Focusing on $P(I|A)$ for the moment, we can compute it using **Bayes' theorem**:

$$P(I|A) = \frac{P(I) \cdot P(A|I)}{P(I) \cdot P(A|I) + P(\neg I) \cdot P(A|\neg I)}$$

$$= \frac{0.00002 \cdot P(A|I)}{0.00002 \cdot P(A|I) + 0.99998 \cdot P(A|\neg I)}$$

Now consider the impact of the false-alarm rate $P(A|\neg I)$ on $P(I|A)$. Even with a very good true-alarm rate of $P(A|I) = 0.8$, a seemingly good false-alarm rate of $P(A|\neg I) = 0.0001$ yields $P(I|A) \approx 0.14$. That is, fewer than one in every seven alarms indicates a real intrusion! In systems where a security administrator investigates each alarm, a high rate of false alarms—called a "Christmas tree effect"—is exceedingly wasteful and will quickly teach the administrator to ignore alarms.

This example illustrates a general principle for IPSs: for usability, they must offer an extremely low false-alarm rate. Achieving a sufficiently low false-alarm rate is an especially serious challenge for anomaly-detection systems, as mentioned, because of the difficulties of adequately benchmarking normal system behavior. However, research continues to improve anomaly-detection techniques. Intrusion-detection software is evolving to implement signatures, anomaly algorithms, and other algorithms and to combine the results to arrive at a more accurate anomaly-detection rate.

### 16.6.4 Virus Protection

As we have seen, viruses can and do wreak havoc on systems. Protection from viruses thus is an important security concern. Antivirus programs are often used to provide this protection. Some of these programs are effective against only particular known viruses. They work by searching all the programs on a system for the specific pattern of instructions known to make up the virus.

When they find a known pattern, they remove the instructions, **disinfecting** the program. Antivirus programs may have catalogs of thousands of viruses for which they search.

Both viruses and antivirus software continue to become more sophisticated. Some viruses modify themselves as they infect other software to avoid the basic pattern-match approach of antivirus programs. Antivirus programs in turn now look for families of patterns rather than a single pattern to identify a virus. In fact, some antivirus programs implement a variety of detection algorithms. They can decompress compressed viruses before checking for a signature. Some also look for process anomalies. A process opening an executable file for writing is suspicious, for example, unless it is a compiler. Another popular technique is to run a program in a **sandbox** (Section 17.11.3), which is a controlled or emulated section of the system. The antivirus software analyzes the behavior of the code in the sandbox before letting it run unmonitored. Some antivirus programs also put up a complete shield rather than just scanning files within a file system. They search boot sectors, memory, inbound and outbound e-mail, files as they are downloaded, files on removable devices or media, and so on.

The best protection against computer viruses is prevention, or the practice of **safe computing**. Purchasing unopened software from vendors and avoiding free or pirated copies from public sources or disk exchange offer the safest route to preventing infection. However, even new copies of legitimate software applications are not immune to virus infection: in a few cases, disgruntled employees of a software company have infected the master copies of software programs to do economic harm to the company. Likewise, hardware devices can come from the factory pre-infected for your convenience. For macro viruses, one defense is to exchange Microsoft Word documents in an alternative file format called **rich text format** (**RTF**). Unlike the native Word format, RTF does not include the capability to attach macros.

Another defense is to avoid opening any e-mail attachments from unknown users. Unfortunately, history has shown that e-mail vulnerabilities appear as fast as they are fixed. For example, in 2000, the ***love bug*** virus became very widespread by traveling in e-mail messages that pretended to be love notes sent by friends of the receivers. Once a receiver opened the attached Visual Basic script, the virus propagated by sending itself to the first addresses in the receiver's e-mail contact list. Fortunately, except for clogging e-mail systems and users' inboxes, it was relatively harmless. It did, however, effectively negate the defensive strategy of opening attachments only from people known to the receiver. A more effective defense method is to avoid opening any e-mail attachment that contains executable code. Some companies now enforce this as policy by removing all incoming attachments to e-mail messages.

Another safeguard, although it does not prevent infection, does permit early detection. A user must begin by completely reformatting the hard disk, especially the boot sector, which is often targeted for viral attack. Only secure software is uploaded, and a signature of each program is taken via a secure message-digest computation. The resulting file name and associated message-digest list must then be kept free from unauthorized access. Periodically, or each time a program is run, the operating system recomputes the signature and compares it with the signature on the original list; any differences serve as a warning of possible infection. This technique can be combined with others. For example, a high-overhead antivirus scan, such as a sandbox, can be used; and if a program passes the test, a signature can be created for it. If the signatures match the next time the program is run, it does not need to be virus-scanned again.

### 16.6.5 Auditing, Accounting, and Logging

Auditing, accounting, and logging can decrease system performance, but they are useful in several areas, including security. Logging can be general or specific. All system-call executions can be logged for analysis of program behavior (or misbehavior). More typically, suspicious events are logged. Authentication failures and authorization failures can tell us quite a lot about break-in attempts.

Accounting is another potential tool in a security administrator's kit. It can be used to find performance changes, which in turn can reveal security problems. One of the early UNIX computer break-ins was detected by Cliff Stoll when he was examining accounting logs and spotted an anomaly.

### 16.6.6 Firewalling to Protect Systems and Networks

We turn next to the question of how a trusted computer can be connected safely to an untrustworthy network. One solution is the use of a firewall to separate trusted and untrusted systems. A **firewal** is a computer, appliance, process, or router that sits between the trusted and the untrusted. A network firewall limits network access between the multiple **security domains** and monitors and logs all connections. It can also limit connections based on source or destination address, source or destination port, or direction of the connection. For instance, web servers use HTTP to communicate with web browsers. A firewall therefore may allow only HTTP to pass from all hosts outside the firewall to the web server within the firewall. The first worm, the Morris Internet worm, used the `finger` protocol to break into computers, so `finger` would not be allowed to pass, for example.

In fact, a network firewall can separate a network into multiple domains. A common implementation has the Internet as the untrusted domain; a semitrusted and semisecure network, called the **demilitarized zone** (**DMZ**), as another domain; and a company's computers as a third domain (Figure 16.10). Connections are allowed from the Internet to the DMZ computers and from the company computers to the Internet but are not allowed from the Internet or DMZ computers to the company computers. Optionally, controlled communications may be allowed between the DMZ and one company computer or more. For instance, a web server on the DMZ may need to query a database server on the corporate network. With a firewall, however, access is contained, and any DMZ systems that are broken into still are unable to access the company computers.

Of course, a firewall itself must be secure and attack-proof. Otherwise, its ability to secure connections can be compromised. Furthermore, firewalls do not prevent attacks that **tunnel**, or travel within protocols or connections

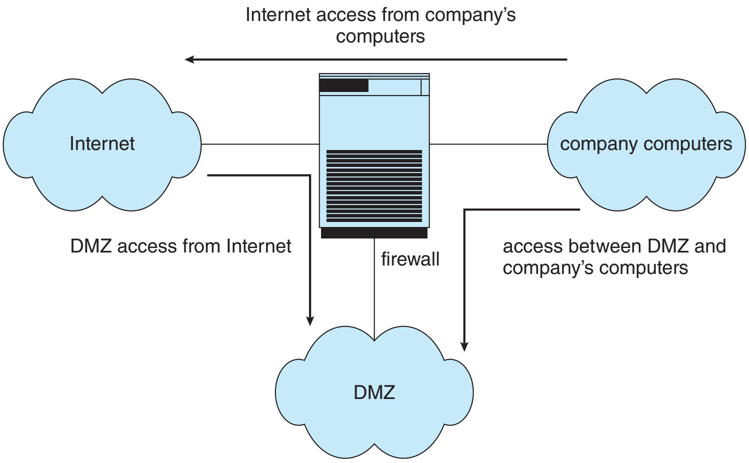

Internet access from company's computers
Internet
company computers
DMZ access from Internet
firewall
access between DMZ and company's computers
DMZ

**Figure 16.10** Domain separation via firewall.

that the firewall allows. A buffer-overflow attack to a web server will not be stopped by the firewall, for example, because the HTTP connection is allowed; it is the contents of the HTTP connection that house the attack. Likewise, denial-of-service attacks can affect firewalls as much as any other machines. Another vulnerability of firewalls is spoofing, in which an unauthorized host pretends to be an authorized host by meeting some authorization criterion. For example, if a firewall rule allows a connection from a host and identifies that host by its IP address, then another host could send packets using that same address and be allowed through the firewall.

In addition to the most common network firewalls, there are other, newer kinds of firewalls, each with its pros and cons. A **personal firewal** is a software layer either included with the operating system or added as an application. Rather than limiting communication between security domains, it limits communication to (and possibly from) a given host. A user could add a personal firewall to her PC so that a Trojan horse would be denied access to the network to which the PC is connected, for example. An **application proxy fire wall** understands the protocols that applications speak across the network. For example, SMTP is used for mail transfer. An application proxy accepts a connection just as an SMTP server would and then initiates a connection to the original destination SMTP server. It can monitor the traffic as it forwards the message, watching for and disabling illegal commands, attempts to exploit bugs, and so on. Some firewalls are designed for one specific protocol. An **XML firewal** , for example, has the specific purpose of analyzing XML traffic and blocking disallowed or malformed XML. **System-call firewalls** sit between applications and the kernel, monitoring system-call execution. For example, in Solaris 10, the "least privilege" feature implements a list of more than fifty system calls that processes may or may not be allowed to make. A process that does not need to spawn other processes can have that ability taken away, for instance.

### 16.6.7 Other Solutions

In the ongoing battle between CPU designers, operating system implementers, and hackers, one particular technique has been helpful to defend against code injection. To mount a code-injection attack, hackers must be able to deduce the exact address in memory of their target. Normally, this may not be difficult, since memory layout tends to be predictable. An operating system technique called **Address Space Layout Randomization** (**ASLR**) attempts to solve this problem by randomizing address spaces—that is, putting address spaces, such as the starting locations of the stack and heap, in unpredictable locations. Address randomization, although not foolproof, makes exploitation considerably more difficult. ASLR is a standard feature in many operating systems, including Windows, Linux, and macOS.

In mobile operating systems such as iOS and Android, an approach often adopted is to place the user data and the system files into two separate partitions. The system partition is mounted read-only, whereas the data partition is read–write. This approach has numerous advantages, not the least of which is greater security: the system partition files cannot easily be tampered with, bolstering system integrity. Android takes this a step further by using Linux's `dm-verity` mechanism to cryptographically hash the system partition and detect any modifications.

### 16.6.8 Security Defenses Summarized

By applying appropriate layers of defense, we can keep systems safe from all but the most persistent attackers. In summary, these layers may include the following:

- Educate users about safe computing—don't attach devices of unknown origin to the computer, don't share passwords, use strong passwords, avoid falling for social engineering appeals, realize that an e-mail is not necessarily a private communication, and so on
- Educate users about how to prevent phishing attacks—don't click on e-mail attachments or links from unknown (or even known) senders; authenticate (for example, via a phone call) that a request is legitimate.
- Use secure communication when possible.
- Physically protect computer hardware.
- Configure the operating system to minimize the attack surface; disable all unused services.
- Configure system daemons, privileges applications, and services to be as secure as possible.
- Use modern hardware and software, as they are likely to have up-to-date security features.
- Keep systems and applications up to date and patched.
- Only run applications from trusted sources (such as those that are code signed).

- Enable logging and auditing; review the logs periodically, or automate alerts.
- Install and use antivirus software on systems susceptible to viruses, and keep the software up to date.
- Use strong passwords and passphrases, and don't record them where they could be found.
- Use intrusion detection, firewalling, and other network-based protection systems as appropriate.
- For important facilities, use periodic vulnerability assessments and other testing methods to test security and response to incidents.
- Encrypt mass-storage devices, and consider encrypting important individual files as well.
- Have a security policy for important systems and facilities, and keep it up to date

## 16.7 An Example: Windows 10

Microsoft Windows 10 is a general-purpose operating system designed to support a variety of security features and methods. In this section, we examine features that Windows 10 uses to perform security functions. For more information and background on Windows, see Appendix B.

The Windows 10 security model is based on the notion of **user accounts**. Windows 10 allows the creation of any number of user accounts, which can be grouped in any manner. Access to system objects can then be permitted or denied as desired. Users are identified to the system by a ***unique*** security ID. When a user logs on, Windows 10 creates a **security access token** that includes the security ID for the user, security IDs for any groups of which the user is a member, and a list of any special privileges that the user has. Examples of special privileges include backing up files and directories, shutting down the computer, logging on interactively, and changing the system clock. Every process that Windows 10 runs on behalf of a user will receive a copy of the access token. The system uses the security IDs in the access token to permit or deny access to system objects whenever the user, or a process on behalf of the user, attempts to access the object. Authentication of a user account is typically accomplished via a user name and password, although the modular design of Windows 10 allows the development of custom authentication packages. For example, a retinal (or eye) scanner might be used to verify that the user is who she says she is.

Windows 10 uses the idea of a subject to ensure that programs run by a user do not get greater access to the system than the user is authorized to have. A **subject** is used to track and manage permissions for each program that a user runs. It is composed of the user's access token and the program acting on behalf of the user. Since Windows 10 operates with a client–server model, two classes of subjects are used to control access: simple subjects and server subjects. An example of a **simple subject** is the typical application program that a user executes after she logs on. The simple subject is assigned a **security**

**context** based on the security access token of the user. A **server subject** is a process implemented as a protected server that uses the security context of the client when acting on the client's behalf.

As mentioned in Section 16.6.6, auditing is a useful security technique. Windows 10 has built-in auditing that allows many common security threats to be monitored. Examples include failure auditing for login and logoff events to detect random password break-ins, success auditing for login and logoff events to detect login activity at strange hours, success and failure write-access auditing for executable files to track a virus outbreak, and success and failure auditing for file access to detect access to sensitive files.

Windows Vista added mandatory integrity control, which works by assigning an **integrity label** to each securable object and subject. In order for a given subject to have access to an object, it must have the access requested in the discretionary access-control list, and its integrity label must be equal to or higher than that of the secured object (for the given operation). The integrity labels in Windows 7 are: untrusted, low, medium, high, and system. In addition, three access mask bits are permitted for integrity labels: NoReadUp, NoWriteUp, and NoExecuteUp. NoWriteUp is automatically enforced, so a lower-integrity subject cannot perform a write operation on a higher-integrity object. However, unless explicitly blocked by the security descriptor, it can perform read or execute operations.

For securable objects without an explicit integrity label, a default label of medium is assigned. The label for a given subject is assigned during logon. For instance, a nonadministrative user will have an integrity label of medium. In addition to integrity labels, Windows Vista also added User Account Control (UAC), which represents an administrative account (not the built-in Administrators account) with two separate tokens. One, for normal usage, has the built-in Administrators group disabled and has an integrity label of medium. The other, for elevated usage, has the built-in Administrators group enabled and an integrity label of high.

Security attributes of an object in Windows 10 are described by a **security descriptor**. The security descriptor contains the security ID of the owner of the object (who can change the access permissions), a group security ID used only by the POSIX subsystem, a discretionary access-control list that identifies which users or groups are allowed (and which are explicitly denied) access, and a system access-control list that controls which auditing messages the system will generate. Optionally, the system access-control list can set the integrity of the object and identify which operations to block from lower-integrity subjects: read, write (always enforced), or execute. For example, the security descriptor of the file `foo.bar` might have owner gwen and this discretionary access-control list:

- owner gwen—all access
- group cs—read–write access
- user maddie—no access

In addition, it might have a system access-control list that tells the system to audit writes by everyone, along with an integrity label of medium that denies read, write, and execute to lower-integrity subjects.

An access-control list is composed of access-control entries that contain the security ID of the individual or group being granted access and an access mask that defines all possible actions on the object, with a value of AccessAllowed or AccessDenied for each action. Files in Windows 10 may have the following access types: `ReadData`, `WriteData`, `AppendData`, `Execute`, `ReadExtendedAttribute`, `WriteExtendedAttribute`, `ReadAttributes`, and `WriteAttributes`. We can see how this allows a fine degree of control over access to objects.

Windows 10 classifies objects as either container objects or noncontainer objects. **Container objects**, such as directories, can logically contain other objects. By default, when an object is created within a container object, the new object inherits permissions from the parent object. Similarly, if the user copies a file from one directory to a new directory, the file will inherit the permissions of the destination directory. **Noncontainer objects** inherit no other permissions. Furthermore, if a permission is changed on a directory, the new permissions do not automatically apply to existing files and subdirectories; the user may explicitly apply them if he so desires.

The system administrator can use the Windows 10 Performance Monitor to help her spot approaching problems. In general, Windows 10 does a good job of providing features to help ensure a secure computing environment. Many of these features are not enabled by default, however, which may be one reason for the myriad security breaches on Windows 10 systems. Another reason is the vast number of services Windows 10 starts at system boot time and the number of applications that typically are installed on a Windows 10 system. For a real multiuser environment, the system administrator should formulate a security plan and implement it, using the features that Windows 10 provides and other security tools.

One feature differentiating security in Windows 10 from earlier versions is code signing. Some versions of Windows 10 make it mandatory—applications that are not properly signed by their authors will not execute—while other versions make it optional or leave it to the administrator to determine what to do with unsigned applications.

## 16.8 Summary

- Protection is an internal problem. Security, in contrast, must consider both the computer system and the environment—people, buildings, businesses, valuable objects, and threats—within which the system is used.

- The data stored in the computer system must be protected from unauthorized access, malicious destruction or alteration, and accidental introduction of inconsistency. It is easier to protect against accidental loss of data consistency than to protect against malicious access to the data. Absolute protection of the information stored in a computer system from malicious abuse is not possible; but the cost to the perpetrator can be made sufficiently high to deter most, if not all, attempts to access that information without proper authority.

- Several types of attacks can be launched against programs and against individual computers or the masses. Stack- and buffer-overflow techniques allow successful attackers to change their level of system access. Viruses and malware require human interaction, while worms are self-perpetuating, sometimes infecting thousands of computers. Denial-of-service attacks prevent legitimate use of target systems.

- Encryption limits the domain of receivers of data, while authentication limits the domain of senders. Encryption is used to provide confidentiality of data being stored or transferred. Symmetric encryption requires a shared key, while asymmetric encryption provides a public key and a private key. Authentication, when combined with hashing, can prove that data have not been changed.
- User authentication methods are used to identify legitimate users of a system. In addition to standard user-name and password protection, several authentication methods are used. One-time passwords, for example, change from session to session to avoid replay attacks. Two-factor authentication requires two forms of authentication, such as a hardware calculator with an activation PIN, or one that presents a different response based on the time. Multifactor authentication uses three or more forms. These methods greatly decrease the chance of authentication forgery.
- Methods of preventing or detecting security incidents include an up-to-date security policy, intrusion-detection systems, antivirus software, auditing and logging of system events, system-call monitoring, code signing, sandboxing, and firewalls.

## Further Reading

Information about viruses and worms can be found at http://www.securelist.com, as well as in [Ludwig (1998)] and [Ludwig (2002)]. Another website containing up-to-date security information is http://www.eeye.com/resources/security-center/research. A paper on the dangers of a computer monoculture can be found at http://cryptome.org/cyberinsecurity.htm.

The first paper discussing least privilege is a Multics overview: https://pdfs.semanticscholar.org/1c8d/06510ad449ad24fbdd164f8008cc730cab47.pdf).

For the original article that explored buffer overflow attacks, see http://phrack.org/issues/49/14.html. For the development version control system `git`, see https://github.com/git/.

[C. Kaufman (2002)] and [Stallings and Brown (2011)] explore the use of cryptography in computer systems. Discussions concerning protection of digital signatures are offered by [Akl (1983)], [Davies (1983)], [Denning (1983)], and [Denning (1984)]. Complete cryptography information is presented in [Schneier (1996)] and [Katz and Lindell (2008)].

Asymmetric key encryption is discussed at https://www-ee.stanford.edu/hellman/publications/24.pdf). The TLS cryptographic protocol is described in detail at https://tools.ietf.org/html/rfc5246. The `nmap` network scanning tool is from http://www.insecure.org/nmap/. For more information on port scans and how they are hidden, see http://phrack.org/issues/49/15.html. Nessus is a commercial vulnerability scanner but can be used for free with limited targets: https://www.tenable.com/products/nessus-home.

## Bibliography

**[Akl (1983)]** S. G. Akl, "Digital Signatures: A Tutorial Survey", *Computer*, Volume 16, Number 2 (1983), pages 15–24.

**[C. Kaufman (2002)]** M. S. C. Kaufman, R. Perlman, *Network Security: Private Communication in a Public World*, Second Edition, Prentice Hall (2002).

**[Davies (1983)]** D. W. Davies, "Applying the RSA Digital Signature to Electronic Mail", *Computer*, Volume 16, Number 2 (1983), pages 55–62.

**[Denning (1983)]** D. E. Denning, "Protecting Public Keys and Signature Keys", *Computer*, Volume 16, Number 2 (1983), pages 27–35.

**[Denning (1984)]** D. E. Denning, "Digital Signatures with RSA and Other Public-Key Cryptosystems", *Communications of the ACM*, Volume 27, Number 4 (1984), pages 388–392.

**[Katz and Lindell (2008)]** J. Katz and Y. Lindell, *Introduction to Modern Cryptography*, Chapman & Hall/CRC Press (2008).

**[Ludwig (1998)]** M. Ludwig, *The Giant Black Book of Computer Viruses*, Second Edition, American Eagle Publications (1998).

**[Ludwig (2002)]** M. Ludwig, *The Little Black Book of Email Viruses*, American Eagle Publications (2002).

**[Schneier (1996)]** B. Schneier, *Applied Cryptography*, Second Edition, John Wiley and Sons (1996).

**[Stallings and Brown (2011)]** W. Stallings and L. Brown, *Computer Security: Principles and Practice*, Second Edition, Prentice Hall (2011).

## Chapter 16 Exercises

**16.1** Buffer-overflow attacks can be avoided by adopting a better programming methodology or by using special hardware support. Discuss these solutions.

**16.2** A password may become known to other users in a variety of ways. Is there a simple method for detecting that such an event has occurred? Explain your answer.

**16.3** What is the purpose of using a “salt” along with a user-provided password? Where should the salt be stored, and how should it be used?

**16.4** The list of all passwords is kept in the operating system. Thus, if a user manages to read this list, password protection is no longer provided. Suggest a scheme that will avoid this problem. (Hint: Use different internal and external representations.)

**16.5** An experimental addition to UNIX allows a user to connect a **watchdog** program to a file. The watchdog is invoked whenever a program requests access to the file. The watchdog then either grants or denies access to the file. Discuss two pros and two cons of using watchdogs for security.

**16.6** Discuss a means by which managers of systems connected to the Internet could design their systems to limit or eliminate the damage done by worms. What are the drawbacks of making the change that you suggest?

**16.7** Make a list of six security concerns for a bank’s computer system. For each item on your list, state whether this concern relates to physical, human, or operating-system security.

**16.8** What are two advantages of encrypting data stored in the computer system?

**16.9** What commonly used computer programs are prone to man-in-the-middle attacks? Discuss solutions for preventing this form of attack.

**16.10** Compare symmetric and asymmetric encryption schemes, and discuss the circumstances under which a distributed system would use one or the other.

**16.11** Why doesn’t $D_{kd,N}(E_{ke,N}(m))$ provide authentication of the sender? To what uses can such an encryption be put?

**16.12** Discuss how the asymmetric encryption algorithm can be used to achieve the following goals.

a. Authentication: the receiver knows that only the sender could have generated the message.

b. Secrecy: only the receiver can decrypt the message.

c. Authentication and secrecy: only the receiver can decrypt the message, and the receiver knows that only the sender could have generated the message.

**16.13** Consider a system that generates 10 million audit records per day. Assume that, on average, there are 10 attacks per day on this system and each attack is reflected in 20 records. If the intrusion-detection system has a true-alarm rate of 0.6 and a false-alarm rate of 0.0005, what percentage of alarms generated by the system corresponds to real intrusions?

**16.14** Mobile operating systems such as iOS and Android place the user data and the system files into two separate partitions. Aside from security, what is an advantage of that separation?

# CHAPTER 17

# Protection

In Chapter 16, we addressed security, which involves guarding computer resources against unauthorized access, malicious destruction or alteration, and accidental introduction of inconsistency. In this chapter, we turn to protection, which involves controlling the access of processes and users to the resources defined by a computer system.

The processes in an operating system must be protected from one another's activities. To provide this protection, we can use various mechanisms to ensure that only processes that have gained proper authorization from the operating system can operate on the files, memory segments, CPU, networking, and other resources of a system. These mechanisms must provide a means for specifying the controls to be imposed, together with a means of enforcement.

### CHAPTER OBJECTIVES

- Discuss the goals and principles of protection in a modern computer system.
- Explain how protection domains, combined with an access matrix, are used to specify the resources a process may access.
- Examine capability- and language-based protection systems.
- Describe how protection mechanisms can mitigate system attacks.

## 17.1 Goals of Protection

As computer systems have become more sophisticated and pervasive in their applications, the need to protect their integrity has also grown. Protection was originally conceived as an adjunct to multiprogramming operating systems, so that untrustworthy users might safely share a common logical name space, such as a directory of files, or a common physical name space, such as memory. Modern protection concepts have evolved to increase the reliability of any complex system that makes use of shared resources and is connected to insecure communications platforms such as the Internet.

We need to provide protection for several reasons. The most obvious is the need to prevent the mischievous, intentional violation of an access restriction by a user. Of more general importance, however, is the need to ensure that each process in a system uses system resources only in ways consistent with stated policies. This requirement is an absolute one for a reliable system.

Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by a malfunctioning subsystem. Also, an unprotected resource cannot defend against use (or misuse) by an unauthorized or incompetent user. A protection-oriented system provides means to distinguish between authorized and unauthorized usage.

The role of protection in a computer system is to provide a mechanism for the enforcement of the policies governing resource use. These policies can be established in a variety of ways. Some are fixed in the design of the system, while others are formulated by the management of a system. Still others are defined by individual users to protect resources they “own.” A protection system, then, must have the flexibility to enforce a variety of policies.

Policies for resource use may vary by application, and they may change over time. For these reasons, protection is no longer the concern solely of the designer of an operating system. The application programmer needs to use protection mechanisms as well, to guard resources created and supported by an application subsystem against misuse. In this chapter, we describe the protection mechanisms the operating system should provide, but application designers can use them as well in designing their own protection software.

Note that ***mechanisms*** are distinct from ***policies.*** Mechanisms determine ***how*** something will be done; policies decide ***what*** will be done. The separation of policy and mechanism is important for flexibility. Policies are likely to change from place to place or time to time. In the worst case, every change in policy would require a change in the underlying mechanism. Using general mechanisms enables us to avoid such a situation.

## 17.2 Principles of Protection

Frequently, a guiding principle can be used throughout a project, such as the design of an operating system. Following this principle simplifies design decisions and keeps the system consistent and easy to understand. A key, time-tested guiding principle for protection is the **principle of least privilege**. As discussed in Chapter 16, this principle dictates that programs, users, and even systems be given just enough privileges to perform their tasks.

Consider one of the tenets of UNIX—that a user should not run as root. (In UNIX, only the root user can execute privileged commands.) Most users innately respect that, fearing an accidental delete operation for which there is no corresponding undelete. Because root is virtually omnipotent, the potential for human error when a user acts as root is grave, and its consequences far reaching.

Now consider that rather than human error, damage may result from malicious attack. A virus launched by an accidental click on an attachment is one example. Another is a buffer overflow or other code-injection attack that is successfully carried out against a root-privileged process (or, in Windows, a process with administrator privileges). Either case could prove catastrophic for the system.

Observing the principle of least privilege would give the system a chance to mitigate the attack—if malicious code cannot obtain root privileges, there is a chance that adequately defined **permissions** may block all, or at least some, of the damaging operations. In this sense, permissions can act like an immune system at the operating-system level.

The principle of least privilege takes many forms, which we examine in more detail later in the chapter. Another important principle, often seen as a derivative of the principle of least privilege, is **compartmentalization**. Compartmentalization is the process of protecting each individual system component through the use of specific permissions and access restrictions. Then, if a component is subverted, another line of defense will "kick in" and keep the attacker from compromising the system any further. Compartmentalization is implemented in many forms—from network demilitarized zones (DMZs) through virtualization.

The careful use of access restrictions can help make a system more secure and can also be beneficial in producing an **audit trail**, which tracks divergences from allowed accesses. An audit trail is a hard record in the system logs. If monitored closely, it can reveal early warnings of an attack or (if its integrity is maintained despite an attack) provide clues as to which attack vectors were used, as well as accurately assess the damage caused.

Perhaps most importantly, no single principle is a panacea for security vulnerabilities. **Defense in depth** must be used: multiple layers of protection should be applied one on top of the other (think of a castle with a garrison, a wall, and a moat to protect it). At the same time, of course, attackers use multiple means to bypass defense in depth, resulting in an ever-escalating arms race.

## 17.3 Protection Rings

As we've seen, the main component of modern operating systems is the kernel, which manages access to system resources and hardware. The kernel, by definition, is a trusted and privileged component and therefore must run with a higher level of privileges than user processes.

To carry out this ***privilege separation,*** hardware support is required. Indeed, all modern hardware supports the notion of separate execution levels, though implementations vary somewhat. A popular model of privilege separation is that of protection rings. In this model, fashioned after Bell–LaPadula (https://www.acsac.org/2005/papers/Bell.pdf), execution is defined as a set of concentric rings, with ring *i* providing a subset of the functionality of ring *j* for any $j < i$. The innermost ring, ring *0*, thus provides the full set of privileges. This pattern is shown in Figure 17.1.

When the system boots, it boots to the highest privilege level. Code at that level performs necessary initialization before dropping to a less privileged level. In order to return to a higher privilege level, code usually calls a special instruction, sometimes referred to as a gate, which provides a portal between rings. The `syscall` instruction (in Intel) is one example. Calling this instruction shifts execution from user to kernel mode. As we have seen, executing a system

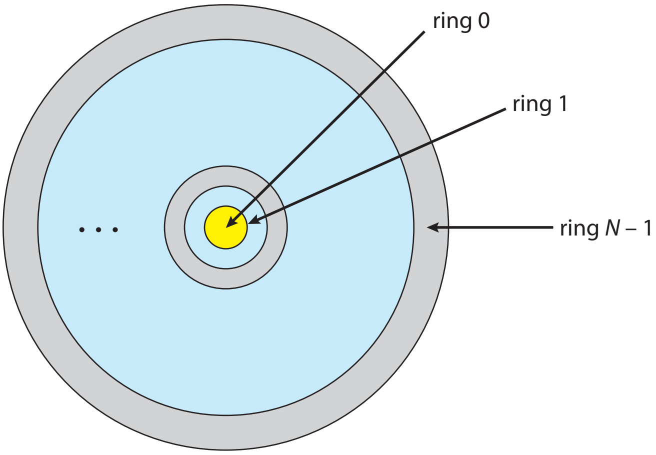


**Figure 17.1** Protection-ring structure.

call will always transfer execution to a predefined address, allowing the caller to specify only arguments (including the system call number), and not arbitrary kernel addresses. In this way, the integrity of the more privileged ring can generally be assured.

Another way of ending up in a more privileged ring is on the occurrence of a processor trap or an interrupt. When either occurs, execution is immediately transferred into the higher-privilege ring. Once again, however, the execution in the higher-privilege ring is predefined and restricted to a well-guarded code path.

Intel architectures follow this model, placing user mode code in ring *3* and kernel mode code in ring *0*. The distinction is made by two bits in the special EFLAGS register. Access to this register is not allowed in ring *3*—thus preventing a malicious process from escalating privileges. With the advent of virtualization, Intel defined an additional ring (*-1*) to allow for **hypervisors**, or virtual machine managers, which create and run virtual machines. Hypervisors have more capabilities than the kernels of the guest operating systems.

The ARM processor's architecture initially allowed only USR and SVC mode, for user and kernel (supervisor) mode, respectively. In ARMv7 processors, ARM introduced **TrustZone** (**TZ**), which provided an additional ring. This most privileged execution environment also has exclusive access to hardware-backed cryptographic features, such as the NFC Secure Element and an on-chip cryptographic key, that make handling passwords and sensitive information more secure. Even the kernel itself has no access to the on-chip key, and it can only request encryption and decryption services from the TrustZone environment (by means of a specialized instruction, **Secure Monitor Call** (**SMC**)), which is only usable from kernel mode. As with system calls, the kernel has no ability to directly execute to specific addresses in the TrustZone—only to pass arguments via registers. Android uses TrustZone extensively as of Version 5.0, as shown in Figure 17.2.

Correctly employing a trusted execution environment means that, if the kernel is compromised, an attacker can't simply retrieve the key from kernel memory. Moving cryptographic services to a separate, trusted environment

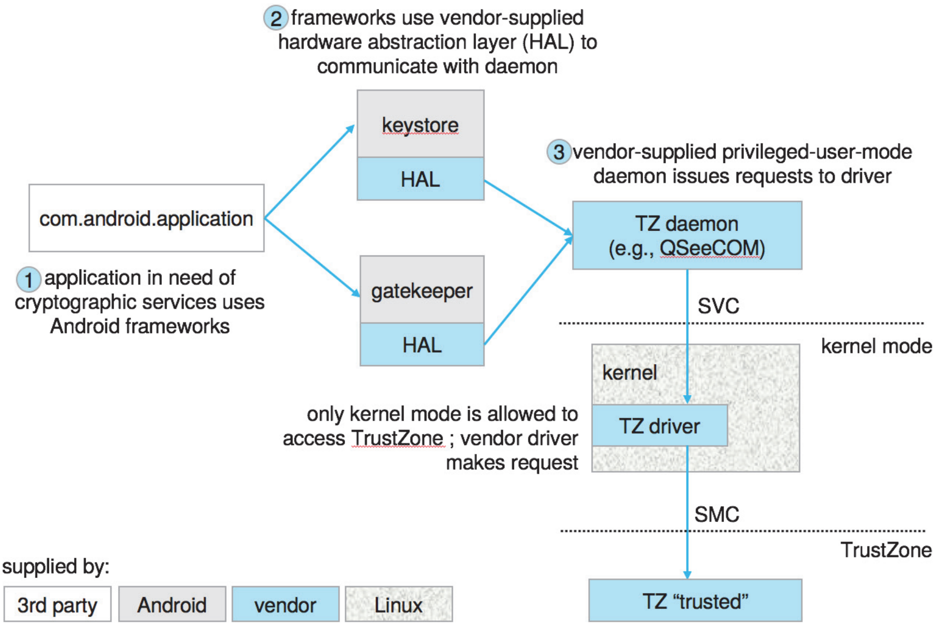

② frameworks use vendor-supplied hardware abstraction layer (HAL) to communicate with daemon
keystore
HAL
gatekeeper
HAL
com.android.application
① application in need of cryptographic services uses Android frameworks
③ vendor-supplied privileged-user-mode daemon issues requests to driver
TZ daemon (e.g., QSeeCOM)
SVC
kernel mode
kernel
TZ driver
only kernel mode is allowed to access TrustZone ; vendor driver makes request
SMC
TrustZone
TZ "trusted"
supplied by:
3rd party
Android
vendor
Linux

**Figure 17.2** Android uses of TrustZone.

also makes brute-force attacks less likely to succeed. (As described in Chapter 16, these attacks involve trying all possible combinations of valid password characters until the password is found.) The various keys used by the system, from the user's password to the system's own, are stored in the on-chip key, which is only accessible in a trusted context. When a key—say, a password—is entered, it is verified via a request to the TrustZone environment. If a key is not known and must be guessed, the TrustZone verifier can impose limitations—by capping the number of verification attempts, for example.

In the 64-bit ARMv8 architecture, ARM extended its model to support four levels, called "exception levels," numbered EL0 through EL3. User mode runs in EL0, and kernel mode in EL1. EL2 is reserved for hypervisors, and EL3 (the most privileged) is reserved for the secure monitor (the TrustZone layer). Any one of the exception levels allows running separate operating systems side by side, as shown in Figure 17.3.

Note that the secure monitor runs at a higher execution level than general-purpose kernels, which makes it the perfect place to deploy code that will check the kernels' integrity. This functionality is included in Samsung's Realtime Kernel Protection (RKP) for Android and Apple's WatchTower (also known as KPP, for Kernel Patch Protection) for iOS.

## 17.4 Domain of Protection

Rings of protection separate functions into domains and order them hierarchically. A generalization of rings is using domains without a hierarchy. A computer system can be treated as a collection of processes and objects. By

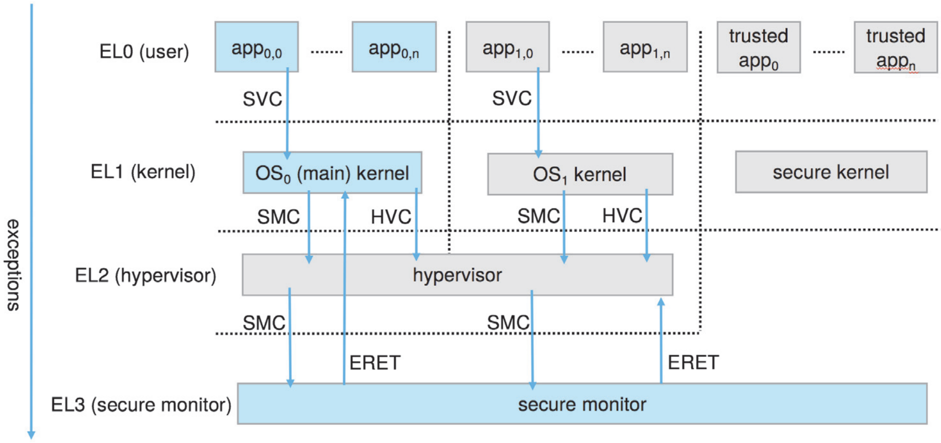


**Figure 17.3** ARM architecture.

***objects,*** we mean both **hardware objects** (such as the CPU, memory segments, printers, disks, and tape drives) and **software objects** (such as files, programs, and semaphores). Each object has a unique name that differentiates it from all other objects in the system, and each can be accessed only through well-defined and meaningful operations. Objects are essentially abstract data types.

The operations that are possible depend on the object. For example, on a CPU, we can only execute. Memory words can be read and written, whereas a DVD-ROM can only be read. Tape drives can be read, written, and rewound. Data files can be created, opened, read, written, closed, and deleted; program files can be read, written, executed, and deleted.

A process should be allowed to access only those objects for which it has authorization. Furthermore, at any time, a process should be able to access only those objects that it currently requires to complete its task. This second requirement, the **need-to-know principle**, is useful in limiting the amount of damage a faulty process or an attacker can cause in the system. For example, when process $p$ invokes procedure `A()`, the procedure should be allowed to access only its own variables and the formal parameters passed to it; it should not be able to access all the variables of process $p$. Similarly, consider the case in which process $p$ invokes a compiler to compile a particular file. The compiler should not be able to access files arbitrarily but should have access only to a well-defined subset of files (such as the source file, output object file, and so on) related to the file to be compiled. Conversely, the compiler may have private files used for accounting or optimization purposes that process $p$ should not be able to access.

In comparing need-to-know with least privilege, it may be easiest to think of need-to-know as the policy and least privilege as the mechanism for achieving this policy. For example, in file permissions, need-to-know might dictate that a user have read access but not write or execute access to a file. The principle of least privilege would require that the operating system provide a mechanism to allow read but not write or execute access.

### 17.4.1 Domain Structure

To facilitate the sort of scheme just described, a process may operate within a **protection domain**, which specifies the resources that the process may access. Each domain defines a set of objects and the types of operations that may be invoked on each object. The ability to execute an operation on an object is an **access right**. A domain is a collection of access rights, each of which is an ordered pair `<object-name, rights-set>`. For example, if domain $D$ has the access right `<file` $F$`,` {read,write}>, then a process executing in domain $D$ can both read and write file $F$. It cannot, however, perform any other operation on that object.

Domains may share access rights. For example, in Figure 17.4, we have three domains: $D_1$, $D_2$, and $D_3$. The access right <$O_4$, {print}> is shared by $D_2$ and $D_3$, implying that a process executing in either of these two domains can print object $O_4$. Note that a process must be executing in domain $D_1$ to read and write object $O_1$, while only processes in domain $D_3$ may execute object $O_1$.

The association between a process and a domain may be either **static**, if the set of resources available to the process is fixed throughout the process's lifetime, or **dynamic**. As might be expected, establishing dynamic protection domains is more complicated than establishing static protection domains.

If the association between processes and domains is fixed, and we want to adhere to the need-to-know principle, then a mechanism must be available to change the content of a domain. The reason stems from the fact that a process may execute in two different phases and may, for example, need read access in one phase and write access in another. If a domain is static, we must define the domain to include both read and write access. However, this arrangement provides more rights than are needed in each of the two phases, since we have read access in the phase where we need only write access, and vice versa. Thus, the need-to-know principle is violated. We must allow the contents of a domain to be modified so that the domain always reflects the minimum necessary access rights.

If the association is dynamic, a mechanism is available to allow **domain switching**, enabling the process to switch from one domain to another. We may also want to allow the content of a domain to be changed. If we cannot change the content of a domain, we can provide the same effect by creating a new domain with the changed content and switching to that new domain when we want to change the domain content.

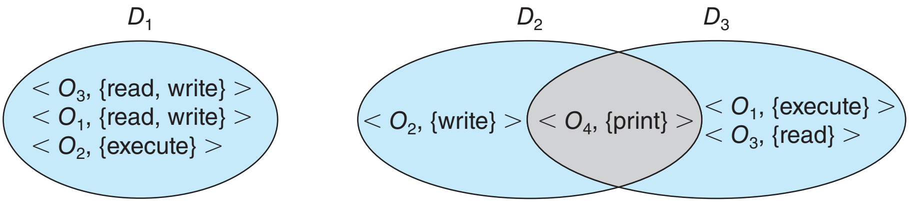


**Figure 17.4** System with three protection domains.

A domain can be realized in a variety of ways:

- Each ***user*** may be a domain. In this case, the set of objects that can be accessed depends on the identity of the user. Domain switching occurs when the user is changed—generally when one user logs out and another user logs in.
- Each ***process*** may be a domain. In this case, the set of objects that can be accessed depends on the identity of the process. Domain switching occurs when one process sends a message to another process and then waits for a response.
- Each ***procedure*** may be a domain. In this case, the set of objects that can be accessed corresponds to the local variables defined within the procedure. Domain switching occurs when a procedure call is made.

We discuss domain switching in greater detail in Section 17.5.

Consider the standard dual-mode (kernel–user mode) model of operating-system execution. When a process is in kernel mode, it can execute privileged instructions and thus gain complete control of the computer system. In contrast, when a process executes in user mode, it can invoke only nonprivileged instructions. Consequently, it can execute only within its predefined memory space. These two modes protect the operating system (executing in kernel domain) from the user processes (executing in user domain). In a multiprogrammed operating system, two protection domains are insufficient, since users also want to be protected from one another. Therefore, a more elaborate scheme is needed. We illustrate such a scheme by examining two influential operating systems—UNIX and Android—to see how they implement these concepts.

### 17.4.2 Example: UNIX

As noted earlier, in UNIX, the root user can execute privileged commands, while other users cannot. Restricting certain operations to the root user can impair other users in their everyday operations, however. Consider, for example, a user who wants to change his password. Inevitably, this requires access to the password database (commonly, `/etc/shadow`), which can only be accessed by root. A similar challenge is encountered when setting a scheduled job (using the `at` command)—doing so requires access to privileged directories that are beyond the reach of a normal user.

The solution to this problem is the setuid bit. In UNIX, an owner identification and a domain bit, known as the ***setuid bit,*** are associated with each file. The setuid bit may or may not be enabled. When the bit is enabled on an executable file (through `chmod +s`), whoever executes the file temporarily assumes the identity of the file owner. That means if a user manages to create a file with the user ID “root” and the setuid bit enabled, anyone who gains access to execute the file becomes user “root” for the duration of the process’s lifetime.

If that strikes you as alarming, it is with good reason. Because of their potential power, setuid executable binaries are expected to be both sterile (affecting only necessary files under specific constraints) and hermetic (for example, tamperproof and impossible to subvert). Setuid programs need to be very carefully written to make these assurances. Returning to the example of changing passwords, the `passwd` command is setuid-root and will indeed modify the password database, but only if first presented with the user's valid password, and it will then restrict itself to editing the password of that user and only that user.

Unfortunately, experience has repeatedly shown that few setuid binaries, if any, fulfill both criteria successfully. Time and again, setuid binaries have been subverted—some through race conditions and others through code injection—yielding instant root access to attackers. Attackers are frequently successful in achieving privilege escalation in this way. Methods of doing so are discussed in Chapter 16. Limiting damage from bugs in setuid programs is discussed in Section 17.8.

### 17.4.3 Example: Android Application IDs

In Android, distinct user IDs are provided on a per-application basis. When an application is installed, the `installd` daemon assigns it a distinct user ID (UID) and group ID (GID), along with a private data directory (`/data/data/<app-name>`) whose ownership is granted to this UID/GID combination alone. In this way, applications on the device enjoy the same level of protection provided by UNIX systems to separate users. This is a quick and simple way to provide isolation, security, and privacy. The mechanism is extended by modifying the kernel to allow certain operations (such as networking sockets) only to members of a particular GID (for example, AID_INET, 3003). A further enhancement by Android is to define certain UIDs as "isolated," which prevents them from initiating RPC requests to any but a bare minimum of services.

## 17.5 Access Matrix

The general model of protection can be viewed abstractly as a matrix, called an **access matrix**. The rows of the access matrix represent domains, and the columns represent objects. Each entry in the matrix consists of a set of access rights. Because the column defines objects explicitly, we can omit the object name from the access right. The entry `access`($i$,$j$) defines the set of operations that a process executing in domain $D_i$ can invoke on object $O_j$.

To illustrate these concepts, we consider the access matrix shown in Figure 17.5. There are four domains and four objects—three files ($F_1$, $F_2$, $F_3$) and one laser printer. A process executing in domain $D_1$ can read files $F_1$ and $F_3$. A process executing in domain $D_4$ has the same privileges as one executing in domain $D_1$; but in addition, it can also write onto files $F_1$ and $F_3$. The laser printer can be accessed only by a process executing in domain $D_2$.

The access-matrix scheme provides us with the mechanism for specifying a variety of policies. The mechanism consists of implementing the access matrix and ensuring that the semantic properties we have outlined hold. More specifically, we must ensure that a process executing in domain $D_i$ can access only those objects specified in row $i$, and then only as allowed by the access-matrix entries.

The access matrix can implement policy decisions concerning protection. The policy decisions involve which rights should be included in the $(i, j)^{th}$ entry.

| object<br>domain | $F_1$ | $F_2$ | $F_3$ | printer |
|---|---|---|---|---|
| $D_1$ | read | | read | |
| $D_2$ | | | | print |
| $D_3$ | | read | execute | |
| $D_4$ | read<br>write | | read<br>write | |

**Figure 17.5** Access matrix.

We must also decide the domain in which each process executes. This last policy is usually decided by the operating system.

The users normally decide the contents of the access-matrix entries. When a user creates a new object $O_j$, the column $O_j$ is added to the access matrix with the appropriate initialization entries, as dictated by the creator. The user may decide to enter some rights in some entries in column $j$ and other rights in other entries, as needed.

The access matrix provides an appropriate mechanism for defining and implementing strict control for both static and dynamic association between processes and domains. When we switch a process from one domain to another, we are executing an operation (`switch`) on an object (the domain). We can control domain switching by including domains among the objects of the access matrix. Similarly, when we change the content of the access matrix, we are performing an operation on an object: the access matrix. Again, we can control these changes by including the access matrix itself as an object. Actually, since each entry in the access matrix can be modified individually, we must consider each entry in the access matrix as an object to be protected. Now, we need to consider only the operations possible on these new objects (domains and the access matrix) and decide how we want processes to be able to execute these operations.

Processes should be able to switch from one domain to another. Switching from domain $D_i$ to domain $D_j$ is allowed if and only if the access right `switch` $\in$ `access`$(i, j)$. Thus, in Figure 17.6, a process executing in domain $D_2$ can switch to domain $D_3$ or to domain $D_4$. A process in domain $D_4$ can switch to $D_1$, and one in domain $D_1$ can switch to $D_2$.

Allowing controlled change in the contents of the access-matrix entries requires three additional operations: `copy`, `owner`, and `control`. We examine these operations next.

The ability to copy an access right from one domain (or row) of the access matrix to another is denoted by an asterisk (*) appended to the access right. The `copy` right allows the access right to be copied only within the column (that is, for the object) for which the right is defined. For example, in Figure 17.7(a), a process executing in domain $D_2$ can copy the read operation into any entry associated with file $F_2$. Hence, the access matrix of Figure 17.7(a) can be modified to the access matrix shown in Figure 17.7(b).

| object<br>domain | $F_1$ | $F_2$ | $F_3$ | laser printer | $D_1$ | $D_2$ | $D_3$ | $D_4$ |
|---|---|---|---|---|---|---|---|---|
| $D_1$ | read | | read | | | switch | | |
| $D_2$ | | | | print | | | switch | switch |
| $D_3$ | | read | execute | | | | | |
| $D_4$ | read<br>write | | read<br>write | | switch | | | |

**Figure 17.6** Access matrix of Figure 17.5 with domains as objects.

This scheme has two additional variants:

1. A right is copied from `access`$(i, j)$ to `access`$(k, j)$; it is then removed from `access`$(i, j)$. This action is a transfer of a right, rather than a copy.
2. Propagation of the `copy` right may be limited. That is, when the right $R^*$ is copied from `access`$(i, j)$ to `access`$(k, j)$, only the right $R$ (not $R^*$) is created. A process executing in domain $D_k$ cannot further copy the right $R$.

A system may select only one of these three `copy` rights, or it may provide all three by identifying them as separate rights: `copy`, `transfer`, and `limited copy`.

| object<br>domain | $F_1$ | $F_2$ | $F_3$ |
|---|---|---|---|
| $D_1$ | execute | | write* |
| $D_2$ | execute | read* | execute |
| $D_3$ | execute | | |

(a)

| object<br>domain | $F_1$ | $F_2$ | $F_3$ |
|---|---|---|---|
| $D_1$ | execute | | write* |
| $D_2$ | execute | read* | execute |
| $D_3$ | execute | read | |

(b)

**Figure 17.7** Access matrix with ***copy*** rights.

We also need a mechanism to allow addition of new rights and removal of some rights. The `owner` right controls these operations. If `access`($i, j$) includes the `owner` right, then a process executing in domain $D_i$ can add and remove any right in any entry in column $j$. For example, in Figure 17.8(a), domain $D_1$ is the owner of $F_1$ and thus can add and delete any valid right in column $F_1$. Similarly, domain $D_2$ is the owner of $F_2$ and $F_3$ and thus can add and remove any valid right within these two columns. Thus, the access matrix of Figure 17.8(a) can be modified to the access matrix shown in Figure 17.8(b).

The `copy` and `owner` rights allow a process to change the entries in a column. A mechanism is also needed to change the entries in a row. The `control` right is applicable only to domain objects. If `access`($i, j$) includes the `control` right, then a process executing in domain $D_i$ can remove any access right from row $j$. For example, suppose that, in Figure 17.6, we include the `control` right in `access`($D_2$, $D_4$). Then, a process executing in domain $D_2$ could modify domain $D_4$, as shown in Figure 17.9.

The `copy` and `owner` rights provide us with a mechanism to limit the propagation of access rights. However, they do not give us the appropriate tools for preventing the propagation (or disclosure) of information. The problem of guaranteeing that no information initially held in an object can migrate outside of its execution environment is called the **confinement problem**. This problem is in general unsolvable (see the bibliographical notes at the end of the chapter).

| object<br>domain | $F_1$ | $F_2$ | $F_3$ |
|---|---|---|---|
| $D_1$ | owner<br>execute |  | write |
| $D_2$ |  | read*<br>owner | read*<br>owner<br>write |
| $D_3$ | execute |  |  |

(a)

| object<br>domain | $F_1$ | $F_2$ | $F_3$ |
|---|---|---|---|
| $D_1$ | owner<br>execute |  | write |
| $D_2$ |  | owner<br>read*<br>write* | read*<br>owner<br>write |
| $D_3$ |  | write | write |

(b)

**Figure 17.8** Access matrix with `owner` rights.

| object<br>domain | $F_1$ | $F_2$ | $F_3$ | laser printer | $D_1$ | $D_2$ | $D_3$ | $D_4$ |
|---|---|---|---|---|---|---|---|---|
| $D_1$ | read | | read | | | switch | | |
| $D_2$ | | | | print | | | switch | switch<br>control |
| $D_3$ | | read | execute | | | | | |
| $D_4$ | write | | write | | switch | | | |

**Figure 17.9** Modified access matrix of Figure 17.6.

These operations on the domains and the access matrix are not in themselves important, but they illustrate the ability of the access-matrix model to let us implement and control dynamic protection requirements. New objects and new domains can be created dynamically and included in the access-matrix model. However, we have shown only that the basic mechanism exists. System designers and users must make the policy decisions concerning which domains are to have access to which objects in which ways.

## 17.6 Implementation of the Access Matrix

How can the access matrix be implemented effectively? In general, the matrix will be sparse; that is, most of the entries will be empty. Although data-structure techniques are available for representing sparse matrices, they are not particularly useful for this application, because of the way in which the protection facility is used. Here, we first describe several methods of implementing the access matrix and then compare the methods.

### 17.6.1 Global Table

The simplest implementation of the access matrix is a global table consisting of a set of ordered triples `<domain, object, rights-set>`. Whenever an operation $M$ is executed on an object $O_j$ within domain $D_i$, the global table is searched for a triple $<D_i, O_j, R_k>$, with $M \in R_k$. If this triple is found, the operation is allowed to continue; otherwise, an exception (or error) condition is raised.

This implementation suffers from several drawbacks. The table is usually large and thus cannot be kept in main memory, so additional I/O is needed. Virtual memory techniques are often used for managing this table. In addition, it is difficult to take advantage of special groupings of objects or domains. For example, if everyone can read a particular object, this object must have a separate entry in every domain.

### 17.6.2 Access Lists for Objects

Each column in the access matrix can be implemented as an access list for one object, as described in Section 13.4.2. Obviously, the empty entries can be discarded. The resulting list for each object consists of ordered pairs `<domain, rights-set>`, which define all domains with a nonempty set of access rights for that object.

This approach can be extended easily to define a list plus a ***default*** set of access rights. When an operation $M$ on an object $O_j$ is attempted in domain $D_i$, we search the access list for object $O_j$, looking for an entry $<D_i, R_k>$ with $M \in R_k$. If the entry is found, we allow the operation; if it is not, we check the default set. If $M$ is in the default set, we allow the access. Otherwise, access is denied, and an exception condition occurs. For efficiency, we may check the default set first and then search the access list.

### 17.6.3 Capability Lists for Domains

Rather than associating the columns of the access matrix with the objects as access lists, we can associate each row with its domain. A **capability list** for a domain is a list of objects together with the operations allowed on those objects. An object is often represented by its physical name or address, called a **capability**. To execute operation $M$ on object $O_j$, the process executes the operation $M$, specifying the capability (or pointer) for object $O_j$ as a parameter. Simple ***possession*** of the capability means that access is allowed.

The capability list is associated with a domain, but it is never directly accessible to a process executing in that domain. Rather, the capability list is itself a protected object, maintained by the operating system and accessed by the user only indirectly. Capability-based protection relies on the fact that the capabilities are never allowed to migrate into any address space directly accessible by a user process (where they could be modified). If all capabilities are secure, the object they protect is also secure against unauthorized access.

Capabilities were originally proposed as a kind of secure pointer, to meet the need for resource protection that was foreseen as multiprogrammed computer systems came of age. The idea of an inherently protected pointer provides a foundation for protection that can be extended up to the application level.

To provide inherent protection, we must distinguish capabilities from other kinds of objects, and they must be interpreted by an abstract machine on which higher-level programs run. Capabilities are usually distinguished from other data in one of two ways:

- Each object has a **tag** to denote whether it is a capability or accessible data. The tags themselves must not be directly accessible by an application program. Hardware or firmware support may be used to enforce this restriction. Although only one bit is necessary to distinguish between capabilities and other objects, more bits are often used. This extension allows all objects to be tagged with their types by the hardware. Thus, the hardware can distinguish integers, floating-point numbers, pointers, Booleans, characters, instructions, capabilities, and uninitialized values by their tags.

- Alternatively, the address space associated with a program can be split into two parts. One part is accessible to the program and contains the program's normal data and instructions. The other part, containing the capability list, is accessible only by the operating system. A segmented memory space is useful to support this approach.

Several capability-based protection systems have been developed; we describe them briefly in Section 17.10. The Mach operating system also uses a version of capability-based protection; it is described in Appendix D.

### 17.6.4 A Lock–Key Mechanism

The **lock–key scheme** is a compromise between access lists and capability lists. Each object has a list of unique bit patterns called **locks**. Similarly, each domain has a list of unique bit patterns called **keys**. A process executing in a domain can access an object only if that domain has a key that matches one of the locks of the object.

As with capability lists, the list of keys for a domain must be managed by the operating system on behalf of the domain. Users are not allowed to examine or modify the list of keys (or locks) directly.

### 17.6.5 Comparison

As you might expect, choosing a technique for implementing an access matrix involves various trade-offs. Using a global table is simple; however, the table can be quite large and often cannot take advantage of special groupings of objects or domains. Access lists correspond directly to the needs of users. When a user creates an object, he can specify which domains can access the object, as well as what operations are allowed. However, because access-right information for a particular domain is not localized, determining the set of access rights for each domain is difficult. In addition, every access to the object must be checked, requiring a search of the access list. In a large system with long access lists, this search can be time consuming.

Capability lists do not correspond directly to the needs of users, but they are useful for localizing information for a given process. The process attempting access must present a capability for that access. Then, the protection system needs only to verify that the capability is valid. Revocation of capabilities, however, may be inefficient (Section 17.7).

The lock–key mechanism, as mentioned, is a compromise between access lists and capability lists. The mechanism can be both effective and flexible, depending on the length of the keys. The keys can be passed freely from domain to domain. In addition, access privileges can be effectively revoked by the simple technique of changing some of the locks associated with the object (Section 17.7).

Most systems use a combination of access lists and capabilities. When a process first tries to access an object, the access list is searched. If access is denied, an exception condition occurs. Otherwise, a capability is created and attached to the process. Additional references use the capability to demonstrate swiftly that access is allowed. After the last access, the capability is destroyed. This strategy was used in the MULTICS system and in the CAL system.

As an example of how such a strategy works, consider a file system in which each file has an associated access list. When a process opens a file, the directory structure is searched to find the file, access permission is checked, and buffers are allocated. All this information is recorded in a new entry in a file table associated with the process. The operation returns an index into this table for the newly opened file. All operations on the file are made by specification of the index into the file table. The entry in the file table then points to the file and its buffers. When the file is closed, the file-table entry is deleted. Since the file table is maintained by the operating system, the user cannot accidentally corrupt it. Thus, the user can access only those files that have been opened. Since access is checked when the file is opened, protection is ensured. This strategy is used in the UNIX system.

The right to access must still be checked on each access, and the file-table entry has a capability only for the allowed operations. If a file is opened for reading, then a capability for read access is placed in the file-table entry. If an attempt is made to write onto the file, the system identifies this protection violation by comparing the requested operation with the capability in the file-table entry.

## 17.7 Revocation of Access Rights

In a dynamic protection system, we may sometimes need to revoke access rights to objects shared by different users. Various questions about revocation may arise:

- **Immediate versus delayed**. Does revocation occur immediately, or is it delayed? If revocation is delayed, can we find out when it will take place?
- **Selective versus general**. When an access right to an object is revoked, does it affect all the users who have an access right to that object, or can we specify a select group of users whose access rights should be revoked?
- **Partial versus total**. Can a subset of the rights associated with an object be revoked, or must we revoke all access rights for this object?
- **Temporary versus permanent**. Can access be revoked permanently (that is, the revoked access right will never again be available), or can access be revoked and later be obtained again?

With an access-list scheme, revocation is easy. The access list is searched for any access rights to be revoked, and they are deleted from the list. Revocation is immediate and can be general or selective, total or partial, and permanent or temporary.

Capabilities, however, present a much more difficult revocation problem, as mentioned earlier. Since the capabilities are distributed throughout the system, we must find them before we can revoke them. Schemes that implement revocation for capabilities include the following:

- **Reacquisition**. Periodically, capabilities are deleted from each domain. If a process wants to use a capability, it may find that that capability has been deleted. The process may then try to reacquire the capability. If access has been revoked, the process will not be able to reacquire the capability.
- **Back-pointers**. A list of pointers is maintained with each object, pointing to all capabilities associated with that object. When revocation is required, we can follow these pointers, changing the capabilities as necessary. This scheme was adopted in the MULTICS system. It is quite general, but its implementation is costly.

- **Indirection**. The capabilities point indirectly, not directly, to the objects. Each capability points to a unique entry in a global table, which in turn points to the object. We implement revocation by searching the global table for the desired entry and deleting it. Then, when an access is attempted, the capability is found to point to an illegal table entry. Table entries can be reused for other capabilities without difficulty, since both the capability and the table entry contain the unique name of the object. The object for a capability and its table entry must match. This scheme was adopted in the CAL system. It does not allow selective revocation.

- **Keys**. A key is a unique bit pattern that can be associated with a capability. This key is defined when the capability is created, and it can be neither modified nor inspected by the process that owns the capability. A **master key** is associated with each object; it can be defined or replaced with the `set-key` operation. When a capability is created, the current value of the master key is associated with the capability. When the capability is exercised, its key is compared with the master key. If the keys match, the operation is allowed to continue; otherwise, an exception condition is raised. Revocation replaces the master key with a new value via the `set-key` operation, invalidating all previous capabilities for this object.

  This scheme does not allow selective revocation, since only one master key is associated with each object. If we associate a list of keys with each object, then selective revocation can be implemented. Finally, we can group all keys into one global table of keys. A capability is valid only if its key matches some key in the global table. We implement revocation by removing the matching key from the table. With this scheme, a key can be associated with several objects, and several keys can be associated with each object, providing maximum flexibility.

  In key-based schemes, the operations of defining keys, inserting them into lists, and deleting them from lists should not be available to all users. In particular, it would be reasonable to allow only the owner of an object to set the keys for that object. This choice, however, is a policy decision that the protection system can implement but should not define.

## 17.8 Role-Based Access Control

In Section 13.4.2, we described how access controls can be used on files within a file system. Each file and directory is assigned an owner, a group, or possibly a list of users, and for each of those entities, access-control information is assigned. A similar function can be added to other aspects of a computer system. A good example of this is found in Solaris 10 and later versions.

The idea is to advance the protection available in the operating system by explicitly adding the principle of least privilege via **role-based access control** (**RBAC**). This facility revolves around privileges. A privilege is the right to execute a system call or to use an option within that system call (such as opening a file with write access). Privileges can be assigned to processes, limiting them to exactly the access they need to perform their work. Privileges and programs can also be assigned to **roles**. Users are assigned roles or can take roles based on passwords assigned to the roles. In this way, a user can take a

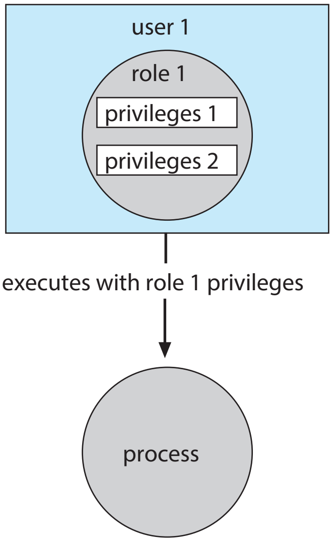


**Figure 17.10** Role-based access control in Solaris 10.

role that enables a privilege, allowing the user to run a program to accomplish a specific task, as depicted in Figure 17.10. This implementation of privileges decreases the security risk associated with superusers and setuid programs.

Notice that this facility is similar to the access matrix described in Section 17.5. This relationship is further explored in the exercises at the end of the chapter.

## 17.9 Mandatory Access Control (MAC)

Operating systems have traditionally used **discretionary access control** (**DAC**) as a means of restricting access to files and other system objects. With DAC, access is controlled based on the identities of individual users or groups. In UNIX-based system, DAC takes the form of file permissions (settable by `chmod`, `chown`, and `chgrp`), whereas Windows (and some UNIX variants) allow finer granularity by means of access-control lists (ACLs).

DACs, however, have proved insufficient over the years. A key weakness lies in their discretionary nature, which allows the owner of a resource to set or modify its permissions. Another weakness is the unlimited access allowed for the administrator or root user. As we have seen, this design can leave the system vulnerable to both accidental and malicious attacks and provides no defense when hackers obtain root privileges.

The need arose, therefore, for a stronger form of protection, which was introduced in the form of **mandatory access control** (**MAC**). MAC is enforced as a system policy that even the root user cannot modify (unless the policy explicitly allows modifications or the system is rebooted, usually into an alternate configuration). The restrictions imposed by MAC policy rules are more powerful than the capabilities of the root user and can be used to make resources inaccessible to anyone but their intended owners.

Modern operating systems all provide MAC along with DAC, although implementations differ. Solaris was among the first to introduce MAC, which was part of Trusted Solaris (2.5). FreeBSD made DAC part of its TrustedBSD implementation (FreeBSD 5.0). The FreeBSD implementation was adopted by Apple in macOS 10.5 and has served as the substrate over which most of the security features of MAC and iOS are implemented. Linux's MAC implementation is part of the SELinux project, which was devised by the NSA, and has been integrated into most distributions. Microsoft Windows joined the trend with Windows Vista's Mandatory Integrity Control.

At the heart of MAC is the concept of **labels**. A label is an identifier (usually a string) assigned to an object (files, devices, and the like). Labels may also be applied to subjects (actors, such as processes). When a subject request to perform operations on the objects. When such requests are to be served by the operating system, it first performs checks defined in a policy, which dictates whether or not a given label holding subject is allowed to perform the operation on the labeled object.

As a brief example, consider a simple set of labels, ordered according to level of privilege: "unclassified," "secret," and "top secret." A user with "secret" clearance will be able to create similarly labeled processes, which will then have access to "unclassified" and "secret" files, but not to "top secret" files. Neither the user nor its processes would even be aware of the existence of "top secret" files, since the operating system would filter them out of all file operations (for example, they would not be displayed when listing directory contents). User processes would similarly be protected themselves in this way, so that an "unclassified" process would not be able to see or perform IPC requests to a "secret" (or "top secret") process. In this way, MAC labels are an implementation of the access matrix described earlier.

## 17.10 Capability-Based Systems

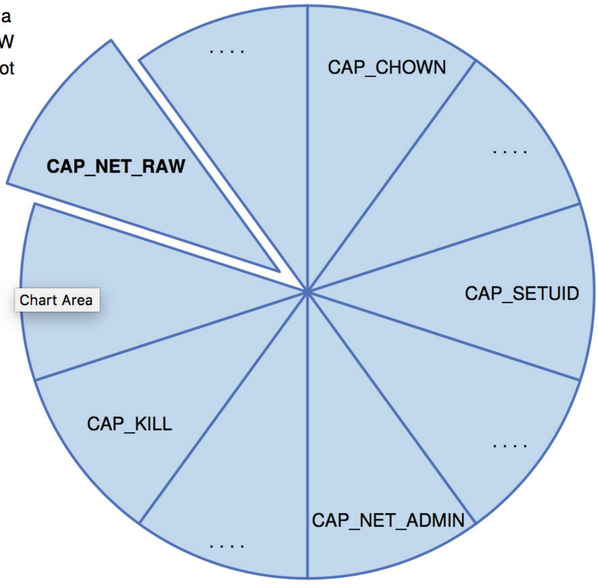
The concept of **capability-based protection** was introduced in the early 1970s. Two early research systems were Hydra and CAP. Neither system was widely used, but both provided interesting proving grounds for protection theories. For more details on these systems, see Section A.14.1 and Section A.14.2. Here, we consider two more contemporary approaches to capabilities.

### 17.10.1 Linux Capabilities

Linux uses capabilities to address the limitations of the UNIX model, which we described earlier. The POSIX standards group introduced capabilities in POSIX 1003.1e. Although POSIX.1e was eventually withdrawn, Linux was quick to adopt capabilities in Version 2.2 and has continued to add new developments.

In essence, Linux's capabilities "slice up" the powers of root into distinct areas, each represented by a bit in a bitmask, as shown in Figure 17.11. Fine-grained control over privileged operations can be achieved by toggling bits in the bitmask.

In practice, three bitmasks are used—denoting the capabilities ***permitted, effective,*** and ***inheritable.*** Bitmasks can apply on a per-process or a per-thread basis. Furthermore, once revoked, capabilities cannot be reacquired. The usual



In the old model, even a simple `ping` utility would have required root privileges, because it opens a raw (ICMP) network socket

Capabilities can be thought of as "slicing up the powers of root" so that individual applications can "cut and choose" only those privileges they actually require

With capabilities, ping can run as a normal user, with CAP_NET_RAW set, allowing it to use ICMP but not other extra privileges

**Figure 17.11** Capabilities in POSIX.1e.

sequence of events is that a process or thread starts with the full set of permitted capabilities and voluntarily decreases that set during execution. For example, after opening a network port, a thread might remove that capability so that no further ports can be opened.

You can probably see that capabilities are a direct implementation of the principle of least privilege. As explained earlier, this tenet of security dictates that an application or user must be given only those rights than are required for its normal operation.

Android (which is based on Linux) also utilizes capabilities, which enable system processes (notably, “system server”), to avoid root ownership, instead selectively enabling only those operations required.

The Linux capabilities model is a great improvement over the traditional UNIX model, but it still is inflexible. For one thing, using a bitmap with a bit representing each capability makes it impossible to add capabilities dynamically and requires recompiling the kernel to add more. In addition, the feature applies only to kernel-enforced capabilities.

### 17.10.2 Darwin Entitlements

Apple’s system protection takes the form of entitlements. Entitlements are declaratory permissions—XML property list stating which permissions are claimed as necessary by the program (see Figure 17.12). When the process attempts a privileged operation (in the figure, loading a kernel extension), its

```
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>com.apple.private.kernel.get-kext-info
  <true/>
  <key>com.apple.rootless.kext-management
  <true/>
</dict>
</plist>
```

**Figure 17.12** Apple Darwin entitlements

entitlements are checked, and only if the needed entitlements are present is the operation allowed.

To prevent programs from arbitrarily claiming an entitlement, Apple embeds the entitlements in the code signature (explained in Section 17.11.4). Once loaded, a process has no way of accessing its code signature. Other processes (and the kernel) can easily query the signature, and in particular the entitlements. Verifying an entitlement is therefore a simple string-matching operation. In this way, only verifiable, authenticated apps may claim entitlements. All system entitlements (`com.apple.*`) are further restricted to Apple's own binaries.

## 17.11 Other Protection Improvement Methods

As the battle to protect systems from accidental and malicious damage escalates, operating-system designers are implementing more types of protection mechanisms at more levels. This section surveys some important real-world protection improvements.

### 17.11.1 System Integrity Protection

Apple introduced in macOS 10.11 a new protection mechanism called **System Integrity Protection** (**SIP**). Darwin-based operating systems use SIP to restrict access to system files and resources in such a way that even the root user cannot tamper with them. SIP uses extended attributes on files to mark them as restricted and further protects system binaries so that they cannot be debugged or scrutinized, much less tampered with. Most importantly, only code-signed kernel extensions are permitted, and SIP can further be configured to allow only code-signed binaries as well.

Under SIP, although root is still the most powerful user in the system, it can do far less than before. The root user can still manage other users' files, as well as install and remove programs, but not in any way that would replace or modify operating-system components. SIP is implemented as a global, inescapable screen on all processes, with the only exceptions allowed for system binaries (for example, `fsck`, or `kextload`, as shown in Figure 17.12), which are specifically entitled for operations for their designated purpose.

### 17.11.2 System-Call Filtering

Recall from Chapter 2 that monolithic systems place all of the functionality of the kernel into a single file that runs in a single address space. Commonly, general-purpose operating-system kernels are monolithic, and they are therefore implicitly trusted as secure. The trust boundary, therefore, rests between kernel mode and user mode—at the system layer. We can reasonably assume that any attempt to compromise the system's integrity will be made from user mode by means of a system call. For example, an attacker can try to gain access by exploiting an unprotected system call.

It is therefore imperative to implement some form of **system-call filtering**. To accomplish this, we can add code to the kernel to perform an inspection at the system-call gate, restricting a caller to a subset of system calls deemed safe or required for that caller's function. Specific system-call profiles can be constructed for individual processes. The Linux mechanism SECCOMP-BPF does just that, harnessing the Berkeley Packet Filter language to load a custom profile through Linux's proprietary `prctl` system call. This filtering is voluntary but can be effectively enforced if called from within a run-time library when it initializes or from within the loader itself before it transfers control to the program's entry point.

A second form of system-call filtering goes deeper still and inspects the arguments of each system call. This form of protection is considered much stronger, as even apparently benign system calls can harbor serious vulnerabilities. This was the case with Linux's fast mutex (`futex`) system call. A race condition in its implementation led to an attacker-controlled kernel memory overwrite and total system compromise. Mutexes are a fundamental component of multitasking, and thus the system call itself could not be filtered out entirely.

A challenge encountered with both approaches is keeping them as flexible as possible while at the same time avoiding the need to rebuild the kernel when changes or new filters are required—a common occurrence due to the differing needs of different processes. Flexibility is especially important given the unpredictable nature of vulnerabilities. New vulnerabilities are discovered every day and may be immediately exploitable by attackers.

One approach to meeting this challenge is to decouple the filter implementation from the kernel itself. The kernel need only contain a set of callouts, which can then be implemented in a specialized driver (Windows), kernel module (Linux), or extension (Darwin). Because an external, modular component provides the filtering logic, it can be updated independently of the kernel. This component commonly makes use of a specialized profiling language by including a built-in interpreter or parser. Thus, the profile itself can be decoupled from the code, providing a human-readable, editable profile and further simplifying updates. It is also possible for the filtering component to call a trusted user-mode daemon process to assist with validation logic.

### 17.11.3 Sandboxing

Sandboxing involves running processes in environments that limit what they can do. In a basic system, a process runs with the credentials of the user that started it and has access to all things that the user can access. If run with system privileges such as root, the process can literally do anything on the system. It is almost always the case that a process does not need full user or system privileges. For example, does a word processor need to accept network connections? Does a network service that provides the time of day need to access files beyond a specific set?

The term **sandboxing** refers to the practice of enforcing strict limitations on a process. Rather than give that process the full set of system calls its privileges would allow, we impose an irremovable set of restrictions on the process in the early stages of its startup—well before the execution of its `main()` function and often as early as its creation with the `fork` system call. The process is then rendered unable to perform any operations outside its allowed set. In this way, it is possible to prevent the process from communicating with any other system component, resulting in tight compartmentalization that mitigates any damage to the system even if the process is compromised.

There are numerous approaches to sandboxing. Java and .net, for example, impose sandbox restrictions at the level of the virtual machine. Other systems enforce sandboxing as part of their mandatory access control (MAC) policy. An example is Android, which draws on an SELinux policy enhanced with specific labels for system properties and service endpoints.

Sandboxing may also be implemented as a combination of multiple mechanisms. Android has found SELinux useful but lacking, because it cannot effectively restrict individual system calls. The latest Android versions (“Nougat” and “O”) use an underlying Linux mechanism called SECCOMP-BPF, mentioned earlier, to apply system-call restrictions through the use of a specialized system call. The C run-time library in Android (“Bionic”) calls this system call to impose restrictions on all Android processes and third-party applications.

Among the major vendors, Apple was the first to implement sandboxing, which appeared in macOS 10.5 (“Tiger”) as “Seatbelt”. Seatbelt was “opt-in” rather than mandatory, allowing but not requiring applications to use it. The Apple sandbox was based on dynamic profiles written in the Scheme language, which provided the ability to control not just which operations were to be allowed or blocked but also their arguments. This capability enabled Apple to create different custom-fit profiles for each binary on the system, a practice that continues to this day. Figure 17.13 depicts a profile example.

Apple’s sandboxing has evolved considerably since its inception. It is now used in the iOS variants, where it serves (along with code signing) as the chief protection against untrusted third-party code. In iOS, and starting with macOS 10.8, the macOS sandbox is mandatory and is automatically enforced for all Mac-store downloaded apps. More recently, as mentioned earlier, Apple adopted the System Integrity Protection (SIP), used in macOS 10.11 and later. SIP is, in effect, a system-wide “platform profile.” Apple enforces it starting at system boot on all processes in the system. Only those processes that are entitled can perform privileged operations, and those are code-signed by Apple and therefore trusted.

```
(version 1)
(deny default)
(allow file-chroot)
(allow file-read-metadata (literal "/var"))
(allow sysctl-read)
(allow mach-per-user-lookup)
(allow mach-lookup)
   (global-name "com.apple.system.logger")
```

**Figure 17.13** A sandbox profile of a MacOS daemon denying most operations.

### 17.11.4 Code Signing

At a fundamental level, how can a system "trust" a program or script? Generally, if the item came as part of the operating system, it should be trusted. But what if the item is changed? If it's changed by a system update, then again it's trustworthy, but otherwise it should not be executable or should require special permission (from the user or administrator) before it is run. Tools from third parties, commercial or otherwise, are more difficult to judge. How can we be sure the tool wasn't modified on its way from where it was created to our systems?

Currently, code signing is the best tool in the protection arsenal for solving these problems. **Code signing** is the digital signing of programs and executables to confirm that they have not been changed since the author created them. It uses a cryptographic hash (Section 16.4.1.3) to test for integrity and authenticity. Code signing is used for operating-system distributions, patches, and third-party tools alike. Some operating systems, including iOS, Windows, and macOS, refuse to run programs that fail their code-signing check. It can also enhance system functionality in other ways. For example, Apple can disable all programs written for a now-obsolete version of iOS by stopping its signing of those programs when they are downloaded from the App Store.

## 17.12 Language-Based Protection

To the degree that protection is provided in computer systems, it is usually achieved through an operating-system kernel, which acts as a security agent to inspect and validate each attempt to access a protected resource. Since comprehensive access validation may be a source of considerable overhead, either we must give it hardware support to reduce the cost of each validation, or we must allow the system designer to compromise the goals of protection. Satisfying all these goals is difficult if the flexibility to implement protection policies is restricted by the support mechanisms provided or if protection environments are made larger than necessary to secure greater operational efficiency.

As operating systems have become more complex, and particularly as they have attempted to provide higher-level user interfaces, the goals of protection have become much more refined. The designers of protection systems have drawn heavily on ideas that originated in programming languages and especially on the concepts of abstract data types and objects. Protection systems are now concerned not only with the identity of a resource to which access is attempted but also with the functional nature of that access. In the newest protection systems, concern for the function to be invoked extends beyond a set of system-defined functions, such as standard file-access methods, to include functions that may be user-defined as well.

Policies for resource use may also vary, depending on the application, and they may be subject to change over time. For these reasons, protection can no longer be considered a matter of concern only to the designer of an operating system. It should also be available as a tool for use by the application designer, so that resources of an application subsystem can be guarded against tampering or the influence of an error.

### 17.12.1 Compiler-Based Enforcement

At this point, programming languages enter the picture. Specifying the desired control of access to a shared resource in a system is making a declarative statement about the resource. This kind of statement can be integrated into a language by an extension of its typing facility. When protection is declared along with data typing, the designer of each subsystem can specify its requirements for protection, as well as its need for use of other resources in a system. Such a specification should be given directly as a program is composed, and in the language in which the program itself is stated. This approach has several significant advantages:

1. Protection needs are simply declared, rather than programmed as a sequence of calls on procedures of an operating system.
2. Protection requirements can be stated independently of the facilities provided by a particular operating system.
3. The means for enforcement need not be provided by the designer of a subsystem.
4. A declarative notation is natural because access privileges are closely related to the linguistic concept of data type.

A variety of techniques can be provided by a programming-language implementation to enforce protection, but any of these must depend on some degree of support from an underlying machine and its operating system. For example, suppose a language is used to generate code to run on the Cambridge CAP system (Section A.14.2). On this system, every storage reference made on the underlying hardware occurs indirectly through a capability. This restriction prevents any process from accessing a resource outside of its protection environment at any time. However, a program may impose arbitrary restrictions on how a resource can be used during execution of a particular code segment. We can implement such restrictions most readily by using the software capabilities provided by CAP. A language implementation might provide standard protected procedures to interpret software capabilities that would realize the protection policies that could be specified in the language. This scheme puts policy specification at the disposal of the programmers, while freeing them from implementing its enforcement.

Even if a system does not provide a protection kernel as powerful as those of Hydra (Section A.14.1) or CAP, mechanisms are still available for implementing protection specifications given in a programming language. The principal distinction is that the ***security*** of this protection will not be as great as that supported by a protection kernel, because the mechanism must rely on more assumptions about the operational state of the system. A compiler can separate references for which it can certify that no protection violation could occur from those for which a violation might be possible, and it can treat them differently. The security provided by this form of protection rests on the assumption that the code generated by the compiler will not be modified prior to or during its execution.

What, then, are the relative merits of enforcement based solely on a kernel, as opposed to enforcement provided largely by a compiler?

- **Security**. Enforcement by a kernel provides a greater degree of security of the protection system itself than does the generation of protection-checking code by a compiler. In a compiler-supported scheme, security rests on correctness of the translator, on some underlying mechanism of storage management that protects the segments from which compiled code is executed, and, ultimately, on the security of files from which a program is loaded. Some of these considerations also apply to a software-supported protection kernel, but to a lesser degree, since the kernel may reside in fixed physical storage segments and may be loaded only from a designated file. With a tagged-capability system, in which all address computation is performed either by hardware or by a fixed microprogram, even greater security is possible. Hardware-supported protection is also relatively immune to protection violations that might occur as a result of either hardware or system software malfunction.

- **Flexibility**. There are limits to the flexibility of a protection kernel in implementing a user-defined policy, although it may supply adequate facilities for the system to provide enforcement of its own policies. With a programming language, protection policy can be declared and enforcement provided as needed by an implementation. If a language does not provide sufficient flexibility, it can be extended or replaced with less disturbance than would be caused by the modification of an operating-system kernel.

- **Efficiency**. The greatest efficiency is obtained when enforcement of protection is supported directly by hardware (or microcode). Insofar as software support is required, language-based enforcement has the advantage that static access enforcement can be verified off-line at compile time. Also, since an intelligent compiler can tailor the enforcement mechanism to meet the specified need, the fixed overhead of kernel calls can often be avoided.

In summary, the specification of protection in a programming language allows the high-level description of policies for the allocation and use of resources. A language implementation can provide software for protection enforcement when automatic hardware-supported checking is unavailable. In addition, it can interpret protection specifications to generate calls on whatever protection system is provided by the hardware and the operating system.

One way of making protection available to the application program is through the use of a software capability that could be used as an object of computation. Inherent in this concept is the idea that certain program components might have the privilege of creating or examining these software capabilities. A capability-creating program would be able to execute a primitive operation that would seal a data structure, rendering the latter's contents inaccessible to any program components that did not hold either the seal or the unseal privilege. Such components might copy the data structure or pass its address to other program components, but they could not gain access to its contents. The reason for introducing such software capabilities is to bring a protection mechanism into the programming language. The only problem with the concept as proposed is that the use of the `seal` and `unseal` operations takes a procedural approach to specifying protection. A nonprocedural or declarative notation seems a preferable way to make protection available to the application programmer.

What is needed is a safe, dynamic access-control mechanism for distributing capabilities to system resources among user processes. To contribute to the overall reliability of a system, the access-control mechanism should be safe to use. To be useful in practice, it should also be reasonably efficient. This requirement has led to the development of a number of language constructs that allow the programmer to declare various restrictions on the use of a specific managed resource. (See the bibliographical notes for appropriate references.) These constructs provide mechanisms for three functions:

1. Distributing capabilities safely and efficiently among customer processes. In particular, mechanisms ensure that a user process will use the managed resource only if it was granted a capability to that resource.
2. Specifying the type of operations that a particular process may invoke on an allocated resource (for example, a reader of a file should be allowed only to read the file, whereas a writer should be able both to read and to write). It should not be necessary to grant the same set of rights to every user process, and it should be impossible for a process to enlarge its set of access rights, except with the authorization of the access-control mechanism.
3. Specifying the order in which a particular process may invoke the various operations of a resource (for example, a file must be opened before it can be read). It should be possible to give two processes different restrictions on the order in which they can invoke the operations of the allocated resource.

The incorporation of protection concepts into programming languages, as a practical tool for system design, is in its infancy. Protection will likely become a matter of greater concern to the designers of new systems with distributed architectures and increasingly stringent requirements on data security. Then the importance of suitable language notations in which to express protection requirements will be recognized more widely.

### 17.12.2 Run-Time-Based Enforcement—Protection in Java

Because Java was designed to run in a distributed environment, the Java virtual machine—or JVM—has many built-in protection mechanisms. Java programs are composed of **classes**, each of which is a collection of data fields and functions (called **methods**) that operate on those fields. The JVM loads a class in response to a request to create instances (or objects) of that class. One of the most novel and useful features of Java is its support for dynamically loading untrusted classes over a network and for executing mutually distrusting classes within the same JVM.

Because of these capabilities, protection is a paramount concern. Classes running in the same JVM may be from different sources and may not be equally trusted. As a result, enforcing protection at the granularity of the JVM process is insufficient. Intuitively, whether a request to open a file should be allowed will generally depend on which class has requested the open. The operating system lacks this knowledge.

Thus, such protection decisions are handled within the JVM. When the JVM loads a class, it assigns the class to a protection domain that gives the permissions of that class. The protection domain to which the class is assigned depends on the URL from which the class was loaded and any digital signatures on the class file. (Digital signatures are covered in Section 16.4.1.3.) A configurable policy file determines the permissions granted to the domain (and its classes). For example, classes loaded from a trusted server might be placed in a protection domain that allows them to access files in the user's home directory, whereas classes loaded from an untrusted server might have no file access permissions at all.

It can be complicated for the JVM to determine what class is responsible for a request to access a protected resource. Accesses are often performed indirectly, through system libraries or other classes. For example, consider a class that is not allowed to open network connections. It could call a system library to request the load of the contents of a URL. The JVM must decide whether or not to open a network connection for this request. But which class should be used to determine if the connection should be allowed, the application or the system library?

The philosophy adopted in Java is to require the library class to explicitly permit a network connection. More generally, in order to access a protected resource, some method in the calling sequence that resulted in the request must explicitly assert the privilege to access the resource. By doing so, this method ***takes responsibility*** for the request. Presumably, it will also perform whatever checks are necessary to ensure the safety of the request. Of course, not every method is allowed to assert a privilege; a method can assert a privilege only if its class is in a protection domain that is itself allowed to exercise the privilege.

This implementation approach is called **stack inspection**. Every thread in the JVM has an associated stack of its ongoing method invocations. When a caller may not be trusted, a method executes an access request within a `doPrivileged` block to perform the access to a protected resource directly or indirectly. `doPrivileged()` is a static method in the `AccessController` class that is passed a class with a `run()` method to invoke. When the `doPrivileged` block is entered, the stack frame for this method is annotated to indicate this fact. Then, the contents of the block are executed. When an access to a protected

| protection domain: | untrusted applet | URL loader | networking |
|---|---|---|---|
| socket permission: | none | *.lucent.com:80, connect | any |
| class: | gui:<br>...<br>get(url);<br>open(addr);<br>... | get(URL u):<br>...<br>doPrivileged {<br>open(‘proxy.lucent.com:80’);<br>}<br><request u from proxy><br>... | open(Addr a):<br>...<br>checkPermission (a, connect);<br>connect (a);<br>... |

**Figure 17.14** Stack inspection.

resource is subsequently requested, either by this method or a method it calls, a call to `checkPermissions()` is used to invoke stack inspection to determine if the request should be allowed. The inspection examines stack frames on the calling thread's stack, starting from the most recently added frame and working toward the oldest. If a stack frame is first found that has the `doPrivileged()` annotation, then `checkPermissions()` returns immediately and silently, allowing the access. If a stack frame is first found for which access is disallowed based on the protection domain of the method's class, then `checkPermissions()` throws an `AccessControlException`. If the stack inspection exhausts the stack without finding either type of frame, then whether access is allowed depends on the implementation (some implementations of the JVM may allow access, while other implementations may not).

Stack inspection is illustrated in Figure 17.14. Here, the `gui()` method of a class in the ***untrusted applet*** protection domain performs two operations, first a `get()` and then an `open()`. The former is an invocation of the `get()` method of a class in the ***URL loader*** protection domain, which is permitted to `open()` sessions to sites in the `lucent.com` domain, in particular a proxy server `proxy.lucent.com` for retrieving URLs. For this reason, the untrusted applet's `get()` invocation will succeed: the `checkPermissions()` call in the networking library encounters the stack frame of the `get()` method, which performed its `open()` in a `doPrivileged` block. However, the untrusted applet's `open()` invocation will result in an exception, because the `checkPermissions()` call finds no `doPrivileged` annotation before encountering the stack frame of the `gui()` method.

Of course, for stack inspection to work, a program must be unable to modify the annotations on its own stack frame or to otherwise manipulate stack inspection. This is one of the most important differences between Java and many other languages (including C++). A Java program cannot directly access memory; it can manipulate only an object for which it has a reference. References cannot be forged, and manipulations are made only through well-defined interfaces. Compliance is enforced through a sophisticated collection of load-time and run-time checks. As a result, an object cannot manipulate its run-time stack, because it cannot get a reference to the stack or other components of the protection system.

More generally, Java's load-time and run-time checks enforce **type safety** of Java classes. Type safety ensures that classes cannot treat integers as pointers, write past the end of an array, or otherwise access memory in arbitrary ways. Rather, a program can access an object only via the methods defined on that object by its class. This is the foundation of Java protection, since it enables a class to effectively **encapsulate** and protect its data and methods from other classes loaded in the same JVM. For example, a variable can be defined as `private` so that only the class that contains it can access it or `protected` so that it can be accessed only by the class that contains it, subclasses of that class, or classes in the same package. Type safety ensures that these restrictions can be enforced.

## 17.13 Summary

- System protection features are guided by the principle of need-to-know and implement mechanisms to enforce the principle of least privilege.
- Computer systems contain objects that must be protected from misuse. Objects may be hardware (such as memory, CPU time, and I/O devices) or software (such as files, programs, and semaphores).
- An access right is permission to perform an operation on an object. A domain is a set of access rights. Processes execute in domains and may use any of the access rights in the domain to access and manipulate objects. During its lifetime, a process may be either bound to a protection domain or allowed to switch from one domain to another.
- A common method of securing objects is to provide a series of protection rings, each with more privileges than the last. ARM, for example, provides four protection levels. The most privileged, TrustZone, is callable only from kernel mode.
- The access matrix is a general model of protection that provides a mechanism for protection without imposing a particular protection policy on the system or its users. The separation of policy and mechanism is an important design property.
- The access matrix is sparse. It is normally implemented either as access lists associated with each object or as capability lists associated with each domain. We can include dynamic protection in the access-matrix model by considering domains and the access matrix itself as objects. Revocation of access rights in a dynamic protection model is typically easier to implement with an access-list scheme than with a capability list.
- Real systems are much more limited than the general model. Older UNIX distributions are representative, providing discretionary access controls of read, write, and execution protection separately for the owner, group, and general public for each file. More modern systems are closer to the general model, or at least provide a variety of protection features to protect the system and its users.
- Solaris 10 and beyond, among other systems, implement the principle of least privilege via role-based access control, a form of access matrix.

Another protection extension is mandatory access control, a form of system policy enforcement.

- Capability-based systems offer finer-grained protection than older models, providing specific abilities to processes by "slicing up" the powers of root into distinct areas. Other methods of improving protection include System Integrity Protection, system-call filtering, sandboxing, and code signing.
- Language-based protection provides finer-grained arbitration of requests and privileges than the operating system is able to provide. For example, a single Java JVM can run several threads, each in a different protection class. It enforces the resource requests through sophisticated stack inspection and via the type safety of the language.

## Further Reading

The concept of a capability evolved from Iliffe's and Jodeit's ***codewords***, which were implemented in the Rice University computer ([Iliffe and Jodeit (1962)]). The term ***capability*** was introduced by [Dennis and Horn (1966)].

The principle of separation of policy and mechanism was advocated by the designer of Hydra ([Levin et al. (1975)]).

The use of minimal operating-system support to enforce protection was advocated by the Exokernel Project ([Ganger et al. (2002)], [Kaashoek et al. (1997)]).

The access-matrix model of protection between domains and objects was developed by [Lampson (1969)] and [Lampson (1971)]. [Popek (1974)] and [Saltzer and Schroeder (1975)] provided excellent surveys on the subject of protection.

The Posix capability standard and the way it was implemented in Linux is described in https://www.usenix.org/legacy/event/usenix03/tech/freenix03/full_papers/gruenbacher/gruenbacher_html/main.html

Details on POSIX.1e and its Linux implementation are provided in https://www.usenix.org/legacy/event/usenix03/tech/freenix03/full_papers/gruenbacher/gruenbacher_html/main.html.

## Bibliography

**[Dennis and Horn (1966)]** J. B. Dennis and E. C. V. Horn, "Programming Semantics for Multiprogrammed Computations", *Communications of the ACM*, Volume 9, Number 3 (1966), pages 143–155.

**[Ganger et al. (2002)]** G. R. Ganger, D. R. Engler, M. F. Kaashoek, H. M. Briceno, R. Hunt, and T. Pinckney, "Fast and Flexible Application-Level Networking on Exokernel Systems", *ACM Transactions on Computer Systems*, Volume 20, Number 1 (2002), pages 49–83.

**[Iliffe and Jodeit (1962)]** J. K. Iliffe and J. G. Jodeit, "A Dynamic Storage Allocation System", *Computer Journal*, Volume 5, Number 3 (1962), pages 200–209.

**[Kaashoek et al. (1997)]** M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie, “Application Performance and Flexibility on Exokernel Systems”, *Proceedings of the ACM Symposium on Operating Systems Principles* (1997), pages 52–65.

**[Lampson (1969)]** B. W. Lampson, “Dynamic Protection Structures”, *Proceedings of the AFIPS Fall Joint Computer Conference* (1969), pages 27–38.

**[Lampson (1971)]** B. W. Lampson, “Protection”, *Proceedings of the Fifth Annual Princeton Conference on Information Systems Science* (1971), pages 437–443.

**[Levin et al. (1975)]** R. Levin, E. S. Cohen, W. M. Corwin, F. J. Pollack, and W. A. Wulf, “Policy/Mechanism Separation in Hydra”, *Proceedings of the ACM Symposium on Operating Systems Principles* (1975), pages 132–140.

**[Popek (1974)]** G. J. Popek, “Protection Structures”, *Computer*, Volume 7, Number 6 (1974), pages 22–33.

**[Saltzer and Schroeder (1975)]** J. H. Saltzer and M. D. Schroeder, “The Protection of Information in Computer Systems”, *Proceedings of the IEEE* (1975), pages 1278–1308.

# Chapter 17 Exercises

**17.11** The access-control matrix can be used to determine whether a process can switch from, say, domain A to domain B and enjoy the access privileges of domain B. Is this approach equivalent to including the access privileges of domain B in those of domain A?

**17.12** Consider a computer system in which computer games can be played by students only between 10 P.M. and 6 A.M., by faculty members between 5 P.M. and 8 A.M., and by the computer center staff at all times. Suggest a scheme for implementing this policy efficiently.

**17.13** What hardware features does a computer system need for efficient capability manipulation? Can these features be used for memory protection?

**17.14** Discuss the strengths and weaknesses of implementing an access matrix using access lists that are associated with objects.

**17.15** Discuss the strengths and weaknesses of implementing an access matrix using capabilities that are associated with domains.

**17.16** Explain why a capability-based system provides greater flexibility than a ring-protection scheme in enforcing protection policies.

**17.17** What is the need-to-know principle? Why is it important for a protection system to adhere to this principle?

**17.18** Discuss which of the following systems allow module designers to enforce the need-to-know principle.

a. Ring-protection scheme

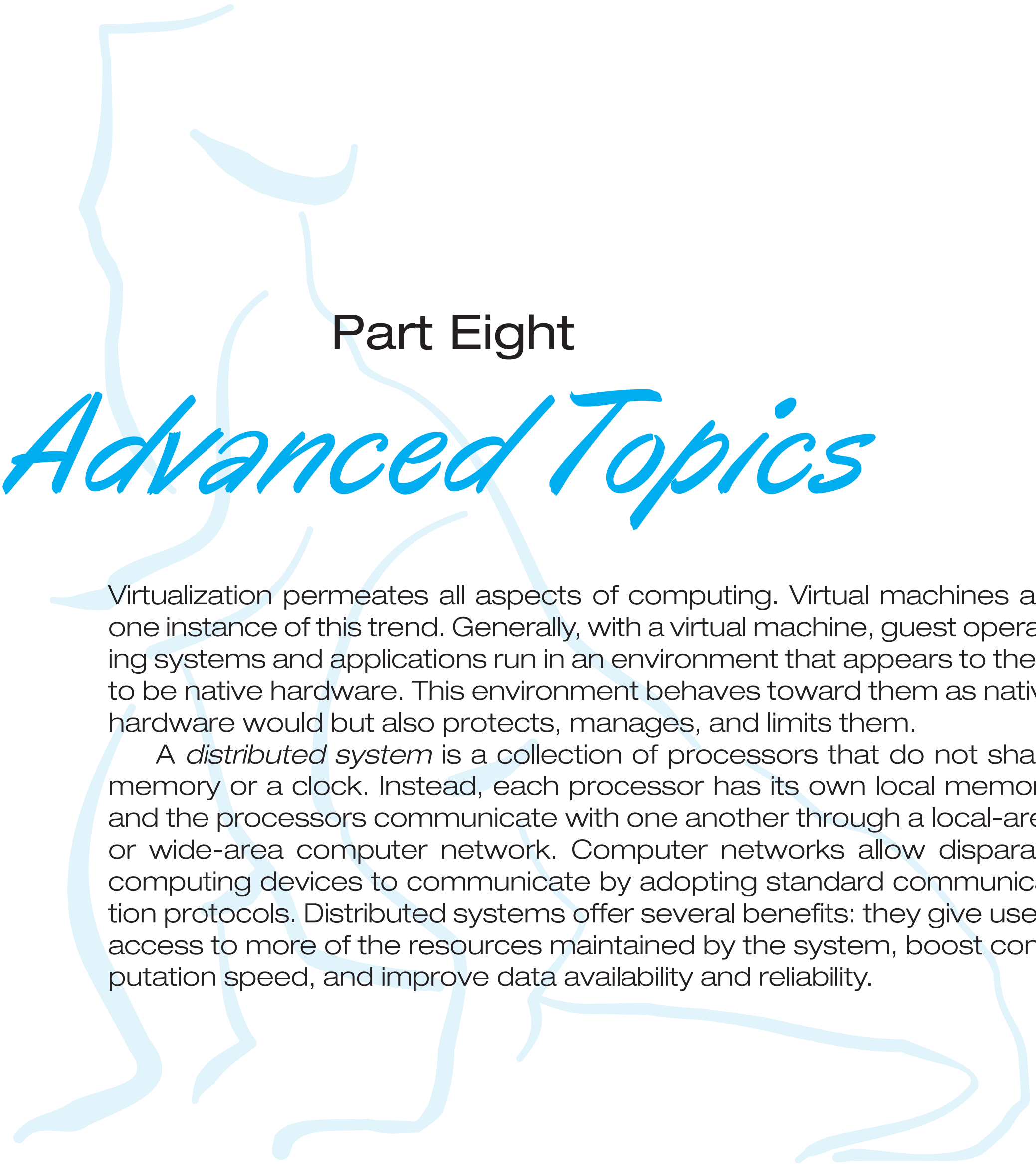
b. JVM's stack-inspection scheme

**17.19** Describe how the Java protection model would be compromised if a Java program were allowed to directly alter the annotations of its stack frame.

**17.20** How are the access-matrix facility and the role-based access-control facility similar? How do they differ?

**17.21** How does the principle of least privilege aid in the creation of protection systems?

**17.22** How can systems that implement the principle of least privilege still have protection failures that lead to security violations?



# Part Eight

# Advanced Topics

Virtualization permeates all aspects of computing. Virtual machines are one instance of this trend. Generally, with a virtual machine, guest operating systems and applications run in an environment that appears to them to be native hardware. This environment behaves toward them as native hardware would but also protects, manages, and limits them.

A *distributed system* is a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory, and the processors communicate with one another through a local-area or wide-area computer network. Computer networks allow disparate computing devices to communicate by adopting standard communication protocols. Distributed systems offer several benefits: they give users access to more of the resources maintained by the system, boost computation speed, and improve data availability and reliability.

# CHAPTER 18

# Virtual Machines

The term ***virtualization*** has many meanings, and aspects of virtualization permeate all aspects of computing. Virtual machines are one instance of this trend. Generally, with a virtual machine, guest operating systems and applications run in an environment that appears to them to be native hardware and that behaves toward them as native hardware would but that also protects, manages, and limits them.

This chapter delves into the uses, features, and implementation of virtual machines. Virtual machines can be implemented in several ways, and this chapter describes these options. One option is to add virtual machine support to the kernel. Because that implementation method is the most pertinent to this book, we explore it most fully. Additionally, hardware features provided by the CPU and even by I/O devices can support virtual machine implementation, so we discuss how those features are used by the appropriate kernel modules.

### CHAPTER OBJECTIVES

- Explore the history and benefits of virtual machines.
- Discuss the various virtual machine technologies.
- Describe the methods used to implement virtualization.
- Identify the most common hardware features that support virtualization and explain how they are used by operating-system modules.
- Discuss current virtualization research areas.

## 18.1 Overview

The fundamental idea behind a virtual machine is to abstract the hardware of a single computer (the CPU, memory, disk drives, network interface cards, and so forth) into several different execution environments, thereby creating the illusion that each separate environment is running on its own private computer. This concept may seem similar to the layered approach of operating system implementation (see Section 2.8.2), and in some ways it is. In the case of virtualization, there is a layer that creates a virtual system on which operating systems or applications can run.

Virtual machine implementations involve several components. At the base is the **host**, the underlying hardware system that runs the virtual machines. The **virtual machine manager** (**VMM**) (also known as a **hypervisor**) creates and runs virtual machines by providing an interface that is ***identical*** to the host (except in the case of paravirtualization, discussed later). Each **guest** process is provided with a virtual copy of the host (Figure 18.1). Usually, the guest process is in fact an operating system. A single physical machine can thus run multiple operating systems concurrently, each in its own virtual machine.

Take a moment to note that with virtualization, the definition of "operating system" once again blurs. For example, consider VMM software such as VMware ESX. This virtualization software is installed on the hardware, runs when the hardware boots, and provides services to applications. The services include traditional ones, such as scheduling and memory management, along with new types, such as migration of applications between systems. Furthermore, the applications are, in fact, guest operating systems. Is the VMware ESX VMM an operating system that, in turn, runs other operating systems? Certainly it acts like an operating system. For clarity, however, we call the component that provides virtual environments a VMM.

The implementation of VMMs varies greatly. Options include the following:

- Hardware-based solutions that provide support for virtual machine creation and management via firmware. These VMMs, which are commonly found in mainframe and large to midsized servers, are generally known as **type 0 hypervisors**. IBM LPARs and Oracle LDOMs are examples.

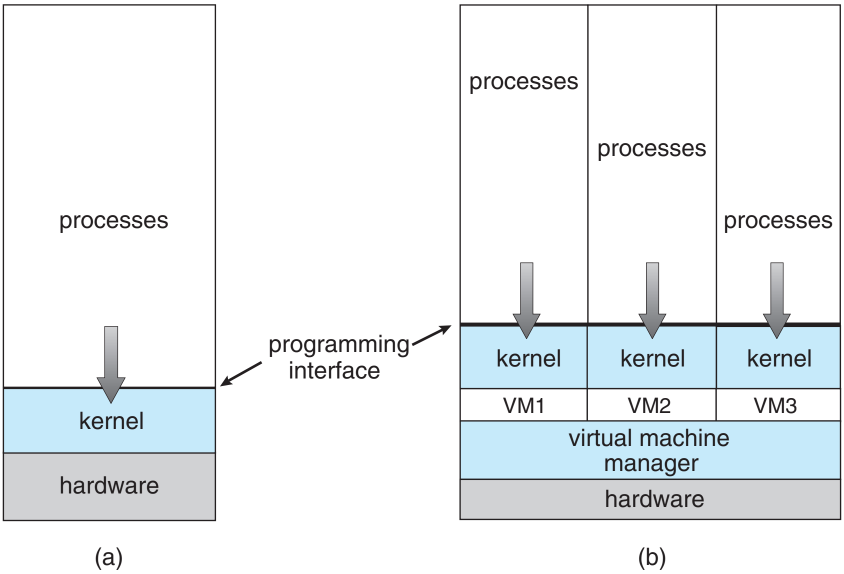


**Figure 18.1** System models. (a) Nonvirtual machine. (b) Virtual machine.

*INDIRECTION*

"All problems in computer science can be solved by another level of indirection"—David Wheeler

". . . except for the problem of too many layers of indirection."—Kevlin Henney

- Operating-system-like software built to provide virtualization, including VMware ESX (mentioned above), Joyent SmartOS, and Citrix XenServer. These VMMs are known as **type 1 hypervisors**.
- General-purpose operating systems that provide standard functions as well as VMM functions, including Microsoft Windows Server with HyperV and Red Hat Linux with the KVM feature. Because such systems have a feature set similar to type 1 hypervisors, they are also known as type 1.
- Applications that run on standard operating systems but provide VMM features to guest operating systems. These applications, which include VMware Workstation and Fusion, Parallels Desktop, and Oracle VirtualBox, are **type 2 hypervisors**.
- **Paravirtualization**, a technique in which the guest operating system is modified to work in cooperation with the VMM to optimize performance.
- **Programming-environment virtualization**, in which VMMs do not virtualize real hardware but instead create an optimized virtual system. This technique is used by Oracle Java and Microsoft.Net.
- **Emulators** that allow applications written for one hardware environment to run on a very different hardware environment, such as a different type of CPU.
- **Application containment**, which is not virtualization at all but rather provides virtualization-like features by segregating applications from the operating system. Oracle Solaris Zones, BSD Jails, and IBM AIX WPARs "contain" applications, making them more secure and manageable.

The variety of virtualization techniques in use today is a testament to the breadth, depth, and importance of virtualization in modern computing. Virtualization is invaluable for data-center operations, efficient application development, and software testing, among many other uses.

## 18.2 History

Virtual machines first appeared commercially on IBM mainframes in 1972. Virtualization was provided by the IBM VM operating system. This system has evolved and is still available. In addition, many of its original concepts are found in other systems, making it worth exploring.

IBM VM/370 divided a mainframe into multiple virtual machines, each running its own operating system. A major difficulty with the VM approach involved disk systems. Suppose that the physical machine had three disk drives but wanted to support seven virtual machines. Clearly, it could not allocate a disk drive to each virtual machine. The solution was to provide virtual disks—termed **minidisks** in IBM's VM operating system. The minidisks were identical to the system's hard disks in all respects except size. The system implemented each minidisk by allocating as many tracks on the physical disks as the minidisk needed.

Once the virtual machines were created, users could run any of the operating systems or software packages that were available on the underlying machine. For the IBM VM system, a user normally ran CMS—a single-user interactive operating system.

For many years after IBM introduced this technology, virtualization remained in its domain. Most systems could not support virtualization. However, a formal definition of virtualization helped to establish system requirements and a target for functionality. The virtualization requirements called for:

- **Fidelity**. A VMM provides an environment for programs that is essentially identical to the original machine.
- **Performance**. Programs running within that environment show only minor performance decreases.
- **Safety**. The VMM is in complete control of system resources.

These requirements still guide virtualization efforts today.

By the late 1990s, Intel 80x86 CPUs had become common, fast, and rich in features. Accordingly, developers launched multiple efforts to implement virtualization on that platform. Both **Xen** and **VMware** created technologies, still used today, to allow guest operating systems to run on the 80x86. Since that time, virtualization has expanded to include all common CPUs, many commercial and open-source tools, and many operating systems. For example, the open-source ***VirtualBox*** project (http://www.virtualbox.org) provides a program that runs on Intel x86 and AMD 64 CPUs and on Windows, Linux, macOS, and Solaris host operating systems. Possible guest operating systems include many versions of Windows, Linux, Solaris, and BSD, including even MS-DOS and IBM OS/2.

## 18.3 Benefits and Features

Several advantages make virtualization attractive. Most of them are fundamentally related to the ability to share the same hardware yet run several different execution environments (that is, different operating systems) concurrently.

One important advantage of virtualization is that the host system is protected from the virtual machines, just as the virtual machines are protected from each other. A virus inside a guest operating system might damage that operating system but is unlikely to affect the host or the other guests. Because each virtual machine is almost completely isolated from all other virtual machines, there are almost no protection problems.

A potential disadvantage of isolation is that it can prevent sharing of resources. Two approaches to providing sharing have been implemented. First, it is possible to share a file-system volume and thus to share files. Second, it is possible to define a network of virtual machines, each of which can send information over the virtual communications network. The network is modeled after physical communication networks but is implemented in software. Of course, the VMM is free to allow any number of its guests to use physical resources, such as a physical network connection (with sharing provided by the VMM), in which case the allowed guests could communicate with each other via the physical network.

One feature common to most virtualization implementations is the ability to freeze, or **suspend**, a running virtual machine. Many operating systems provide that basic feature for processes, but VMMs go one step further and allow copies and **snapshots** to be made of the guest. The copy can be used to create a new VM or to move a VM from one machine to another with its current state intact. The guest can then **resume** where it was, as if on its original machine, creating a **clone**. The snapshot records a point in time, and the guest can be reset to that point if necessary (for example, if a change was made but is no longer wanted). Often, VMMs allow many snapshots to be taken. For example, snapshots might record a guest's state every day for a month, making restoration to any of those snapshot states possible. These abilities are used to good advantage in virtual environments.

A virtual machine system is a perfect vehicle for operating-system research and development. Normally, changing an operating system is a difficult task. Operating systems are large and complex programs, and a change in one part may cause obscure bugs to appear in some other part. The power of the operating system makes changing it particularly dangerous. Because the operating system executes in kernel mode, a wrong change in a pointer could cause an error that would destroy the entire file system. Thus, it is necessary to test all changes to the operating system carefully.

Of course, the operating system runs on and controls the entire machine, so the system must be stopped and taken out of use while changes are made and tested. This period is commonly called **system-development time**. Since it makes the system unavailable to users, system-development time on shared systems is often scheduled late at night or on weekends, when system load is low.

A virtual-machine system can eliminate much of this latter problem. System programmers are given their own virtual machine, and system development is done on the virtual machine instead of on a physical machine. Normal system operation is disrupted only when a completed and tested change is ready to be put into production.

Another advantage of virtual machines for developers is that multiple operating systems can run concurrently on the developer's workstation. This virtualized workstation allows for rapid porting and testing of programs in varying environments. In addition, multiple versions of a program can run, each in its own isolated operating system, within one system. Similarly, quality-assurance engineers can test their applications in multiple environments without buying, powering, and maintaining a computer for each environment.

A major advantage of virtual machines in production data-center use is system **consolidation**, which involves taking two or more separate systems and running them in virtual machines on one system. Such physical-to-virtual conversions result in resource optimization, since many lightly used systems can be combined to create one more heavily used system.

Consider, too, that management tools that are part of the VMM allow system administrators to manage many more systems than they otherwise could. A virtual environment might include 100 physical servers, each running 20 virtual servers. Without virtualization, 2,000 servers would require several system administrators. With virtualization and its tools, the same work can be managed by one or two administrators. One of the tools that make this possible is **templating**, in which one standard virtual machine image, including an installed and configured guest operating system and applications, is saved and used as a source for multiple running VMs. Other features include managing the patching of all guests, backing up and restoring the guests, and monitoring their resource use.

Virtualization can improve not only resource utilization but also resource management. Some VMMs include a **live migration** feature that moves a running guest from one physical server to another without interrupting its operation or active network connections. If a server is overloaded, live migration can thus free resources on the source host while not disrupting the guest. Similarly, when host hardware must be repaired or upgraded, guests can be migrated to other servers, the evacuated host can be maintained, and then the guests can be migrated back. This operation occurs without downtime and without interruption to users.

Think about the possible effects of virtualization on how applications are deployed. If a system can easily add, remove, and move a virtual machine, then why install applications on that system directly? Instead, the application could be preinstalled on a tuned and customized operating system in a virtual machine. This method would offer several benefits for application developers. Application management would become easier, less tuning would be required, and technical support of the application would be more straightforward. System administrators would find the environment easier to manage as well. Installation would be simple, and redeploying the application to another system would be much easier than the usual steps of uninstalling and reinstalling. For widespread adoption of this methodology to occur, though, the format of virtual machines must be standardized so that any virtual machine will run on any virtualization platform. The "Open Virtual Machine Format" is an attempt to provide such standardization, and it could succeed in unifying virtual machine formats.

Virtualization has laid the foundation for many other advances in computer facility implementation, management, and monitoring. **Cloud computing**, for example, is made possible by virtualization in which resources such as CPU, memory, and I/O are provided as services to customers using Internet technologies. By using APIs, a program can tell a cloud computing facility to create thousands of VMs, all running a specific guest operating system and application, that others can access via the Internet. Many multiuser games, photo-sharing sites, and other web services use this functionality.

In the area of desktop computing, virtualization is enabling desktop and laptop computer users to connect remotely to virtual machines located in remote data centers and access their applications as if they were local. This practice can increase security, because no data are stored on local disks at the user's site. The cost of the user's computing resource may also decrease. The user must have networking, CPU, and some memory, but all that these system components need to do is display an image of the guest as its runs remotely (via a protocol such as **RDP**). Thus, they need not be expensive, high-performance components. Other uses of virtualization are sure to follow as it becomes more prevalent and hardware support continues to improve.

## 18.4 Building Blocks

Although the virtual machine concept is useful, it is difficult to implement. Much work is required to provide an ***exact*** duplicate of the underlying machine. This is especially a challenge on dual-mode systems, where the underlying machine has only user mode and kernel mode. In this section, we examine the building blocks that are needed for efficient virtualization. Note that these building blocks are not required by type 0 hypervisors, as discussed in Section 18.5.2.

The ability to virtualize depends on the features provided by the CPU. If the features are sufficient, then it is possible to write a VMM that provides a guest environment. Otherwise, virtualization is impossible. VMMs use several techniques to implement virtualization, including trap-and-emulate and binary translation. We discuss each of these techniques in this section, along with the hardware support needed to support virtualization.

As you read the section, keep in mind that an important concept found in most virtualization options is the implementation of a **virtual CPU** (**VCPU**). The VCPU does not execute code. Rather, it represents the state of the CPU as the guest machine believes it to be. For each guest, the VMM maintains a VCPU representing that guest's current CPU state. When the guest is context-switched onto a CPU by the VMM, information from the VCPU is used to load the right context, much as a general-purpose operating system would use the PCB.

### 18.4.1 Trap-and-Emulate

On a typical dual-mode system, the virtual machine guest can execute only in user mode (unless extra hardware support is provided). The kernel, of course, runs in kernel mode, and it is not safe to allow user-level code to run in kernel mode. Just as the physical machine has two modes, so must the virtual machine. Consequently, we must have a virtual user mode and a virtual kernel mode, both of which run in physical user mode. Those actions that cause a transfer from user mode to kernel mode on a real machine (such as a system call, an interrupt, or an attempt to execute a privileged instruction) must also cause a transfer from virtual user mode to virtual kernel mode in the virtual machine.

How can such a transfer be accomplished? The procedure is as follows: When the kernel in the guest attempts to execute a privileged instruction, that is an error (because the system is in user mode) and causes a trap to the VMM in the real machine. The VMM gains control and executes (or "emulates") the action that was attempted by the guest kernel on the part of the guest. It

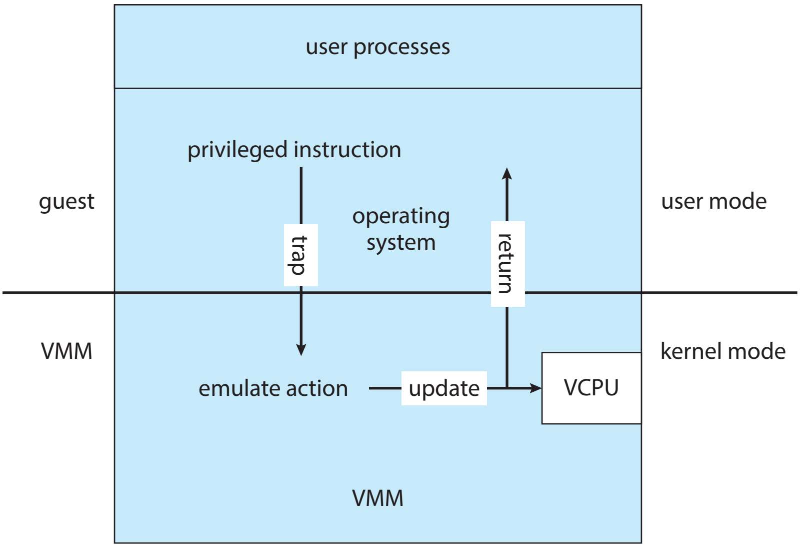


**Figure 18.2** Trap-and-emulate virtualization implementation.

then returns control to the virtual machine. This is called the **trap-and-emulate** method and is shown in Figure 18.2.

With privileged instructions, time becomes an issue. All nonprivileged instructions run natively on the hardware, providing the same performance for guests as native applications. Privileged instructions create extra overhead, however, causing the guest to run more slowly than it would natively. In addition, the CPU is being multiprogrammed among many virtual machines, which can further slow down the virtual machines in unpredictable ways.

This problem has been approached in various ways. IBM VM, for example, allows normal instructions for the virtual machines to execute directly on the hardware. Only the privileged instructions (needed mainly for I/O) must be emulated and hence execute more slowly. In general, with the evolution of hardware, the performance of trap-and-emulate functionality has been improved, and cases in which it is needed have been reduced. For example, many CPUs now have extra modes added to their standard dual-mode operation. The VCPU need not keep track of what mode the guest operating system is in, because the physical CPU performs that function. In fact, some CPUs provide guest CPU state management in hardware, so the VMM need not supply that functionality, removing the extra overhead.

### 18.4.2 Binary Translation

Some CPUs do not have a clean separation of privileged and nonprivileged instructions. Unfortunately for virtualization implementers, the Intel x86 CPU line is one of them. No thought was given to running virtualization on the x86 when it was designed. (In fact, the first CPU in the family—the Intel 4004, released in 1971—was designed to be the core of a calculator.) The chip has maintained backward compatibility throughout its lifetime, preventing changes that would have made virtualization easier through many generations.

Let's consider an example of the problem. The command `popf` loads the flag register from the contents of the stack. If the CPU is in privileged mode, all of the flags are replaced from the stack. If the CPU is in user mode, then only some flags are replaced, and others are ignored. Because no trap is generated if `popf` is executed in user mode, the trap-and-emulate procedure is rendered useless. Other x86 instructions cause similar problems. For the purposes of this discussion, we will call this set of instructions ***special instructions.*** As recently as 1998, using the trap-and-emulate method to implement virtualization on the x86 was considered impossible because of these special instructions.

This previously insurmountable problem was solved with the implementation of the **binary translation** technique. Binary translation is fairly simple in concept but complex in implementation. The basic steps are as follows:

1. If the guest VCPU is in user mode, the guest can run its instructions natively on a physical CPU.
2. If the guest VCPU is in kernel mode, then the guest believes that it is running in kernel mode. The VMM examines every instruction the guest executes in virtual kernel mode by reading the next few instructions that the guest is going to execute, based on the guest's program counter. Instructions other than special instructions are run natively. Special instructions are translated into a new set of instructions that perform the equivalent task—for example, changing the flags in the VCPU.

Binary translation is shown in Figure 18.3. It is implemented by translation code within the VMM. The code reads native binary instructions dynamically from the guest, on demand, and generates native binary code that executes in place of the original code.

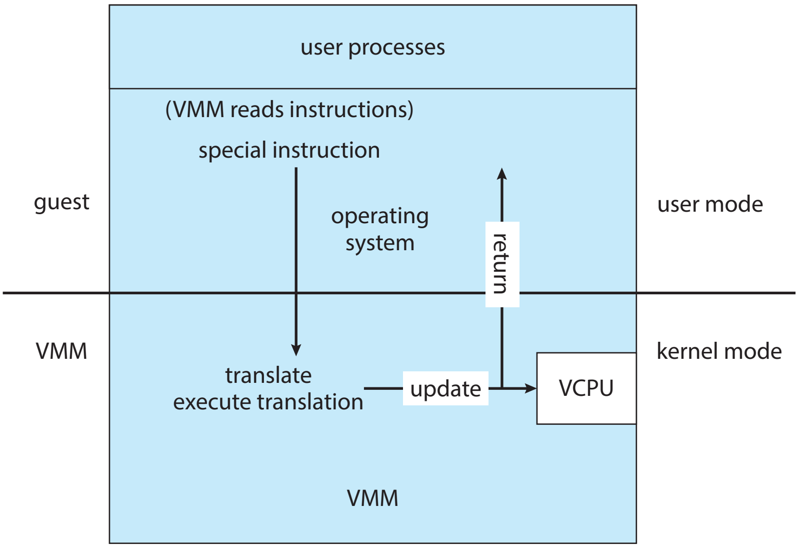


**Figure 18.3** Binary translation virtualization implementation.

The basic method of binary translation just described would execute correctly but perform poorly. Fortunately, the vast majority of instructions would execute natively. But how could performance be improved for the other instructions? We can turn to a specific implementation of binary translation, the VMware method, to see one way of improving performance. Here, caching provides the solution. The replacement code for each instruction that needs to be translated is cached. All later executions of that instruction run from the translation cache and need not be translated again. If the cache is large enough, this method can greatly improve performance.

Let's consider another issue in virtualization: memory management, specifically the page tables. How can the VMM keep page-table state both for guests that believe they are managing the page tables and for the VMM itself? A common method, used with both trap-and-emulate and binary translation, is to use **nested page tables** (**NPTs**). Each guest operating system maintains one or more page tables to translate from virtual to physical memory. The VMM maintains NPTs to represent the guest's page-table state, just as it creates a VCPU to represent the guest's CPU state. The VMM knows when the guest tries to change its page table, and it makes the equivalent change in the NPT. When the guest is on the CPU, the VMM puts the pointer to the appropriate NPT into the appropriate CPU register to make that table the active page table. If the guest needs to modify the page table (for example, fulfilling a page fault), then that operation must be intercepted by the VMM and appropriate changes made to the nested and system page tables. Unfortunately, the use of NPTs can cause TLB misses to increase, and many other complexities need to be addressed to achieve reasonable performance.

Although it might seem that the binary translation method creates large amounts of overhead, it performed well enough to launch a new industry aimed at virtualizing Intel x86-based systems. VMware tested the performance impact of binary translation by booting one such system, Windows XP, and immediately shutting it down while monitoring the elapsed time and the number of translations produced by the binary translation method. The result was 950,000 translations, taking 3 microseconds each, for a total increase of 3 seconds (about 5 percent) over native execution of Windows XP. To achieve that result, developers used many performance improvements that we do not discuss here. For more information, consult the bibliographical notes at the end of this chapter.

### 18.4.3 Hardware Assistance

Without some level of hardware support, virtualization would be impossible. The more hardware support available within a system, the more feature-rich and stable the virtual machines can be and the better they can perform. In the Intel x86 CPU family, Intel added new virtualization support (the **VT-x** instructions) in successive generations beginning in 2005. Now, binary translation is no longer needed.

In fact, all major general-purpose CPUs now provide extended hardware support for virtualization. For example, AMD virtualization technology (**AMD-V**) has appeared in several AMD processors starting in 2006. It defines two new modes of operation—host and guest—thus moving from a dual-mode to a multimode processor. The VMM can enable host mode, define the characteristics of each guest virtual machine, and then switch the system to guest mode, passing control of the system to a guest operating system that is running in the virtual machine. In guest mode, the virtualized operating system thinks it is running on native hardware and sees whatever devices are included in the host's definition of the guest. If the guest tries to access a virtualized resource, then control is passed to the VMM to manage that interaction. The functionality in Intel VT-x is similar, providing root and nonroot modes, equivalent to host and guest modes. Both provide guest VCPU state data structures to load and save guest CPU state automatically during guest context switches. In addition, **virtual machine control structures** (**VMCSs**) are provided to manage guest and host state, as well as various guest execution controls, exit controls, and information about why guests exit back to the host. In the latter case, for example, a nested page-table violation caused by an attempt to access unavailable memory can result in the guest's exit.

AMD and Intel have also addressed memory management in the virtual environment. With AMD's RVI and Intel's EPT memory-management enhancements, VMMs no longer need to implement software NPTs. In essence, these CPUs implement nested page tables in hardware to allow the VMM to fully control paging while the CPUs accelerate the translation from virtual to physical addresses. The NPTs add a new layer, one representing the guest's view of logical-to-physical address translation. The CPU page-table walking function (traversing the data structure to find the desired data) includes this new layer as necessary, walking through the guest table to the VMM table to find the physical address desired. A TLB miss results in a performance penalty, because more tables (the guest and host page tables) must be traversed to complete the lookup. Figure 18.4 shows the extra translation work performed by the hardware to translate from a guest virtual address to a final physical address.

I/O is another area improved by hardware assistance. Consider that the standard direct-memory-access (DMA) controller accepts a target memory address and a source I/O device and transfers data between the two without operating-system action. Without hardware assistance, a guest might try to set up a DMA transfer that affects the memory of the VMM or other guests. In CPUs that provide hardware-assisted DMA (such as Intel CPUs with VT-d), even DMA has a level of indirection. First, the VMM sets up **protection domains** to tell the CPU which physical memory belongs to each guest. Next, it assigns the I/O devices to the protection domains, allowing them direct access to those memory regions and only those regions. The hardware then transforms the address in a DMA request issued by an I/O device to the host physical memory address associated with the I/O. In this manner, DMA transfers are passed through between a guest and a device without VMM interference.

Similarly, interrupts must be delivered to the appropriate guest and must not be visible to other guests. By providing an interrupt remapping feature, CPUs with virtualization hardware assistance automatically deliver an interrupt destined for a guest to a core that is currently running a thread of that guest. That way, the guest receives interrupts without any need for the VMM to intercede in their delivery. Without interrupt remapping, malicious guests could generate interrupts that could be used to gain control of the host system. (See the bibliographical notes at the end of this chapter for more details.)

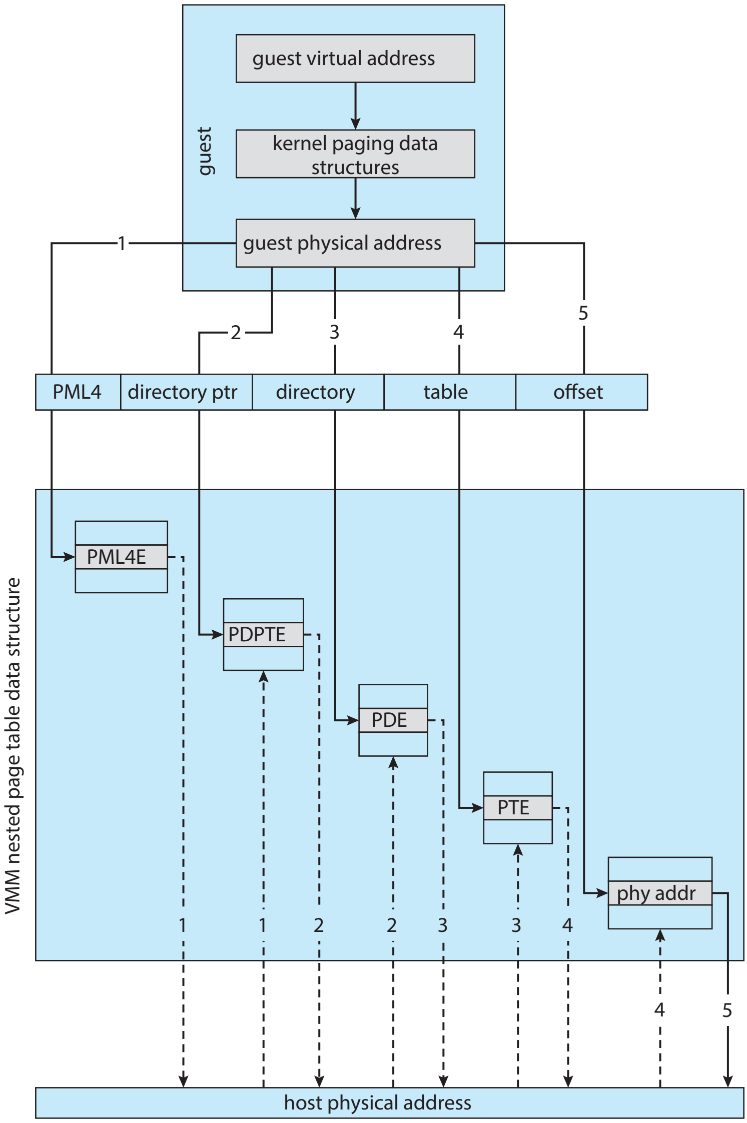

guest
guest virtual address
kernel paging data structures
guest physical address
1
2
3
4
5
PML4
directory ptr
directory
table
offset
VMM nested page table data structure
PML4E
PDPTE
PDE
PTE
phy addr
1
1
2
2
3
3
4
4
5
host physical address

**Figure 18.4** Nested page tables.

ARM architectures, specifically ARM v8 (64-bit) take a slightly different approach to hardware support of virtualization. They provide an entire exception level—EL2—which is even more privileged than that of the kernel (EL1). This allows the running of a secluded hypervisor, with its own MMU access and interrupt trapping. To allow for paravirtualization, a special instruction (HVC) is added. It allows the hypervisor to be called from guest kernels. This instruction can only be called from within kernel mode (EL1).

An interesting side effect of hardware-assisted virtualization is that it allows for the creation of thin hypervisors. A good example is macOS's hypervisor framework ("HyperVisor.framework"), which is an operating-system-supplied library that allows the creation of virtual machines in a few lines of code. The actual work is done via system calls, which have the kernel call the privileged virtualization CPU instructions on behalf of the hypervisor process, allowing management of virtual machines without the hypervisor needing to load a kernel module of its own to execute those calls.

## 18.5 Types of VMs and Their Implementations

We've now looked at some of the techniques used to implement virtualization. Next, we consider the major types of virtual machines, their implementation, their functionality, and how they use the building blocks just described to create a virtual environment. Of course, the hardware on which the virtual machines are running can cause great variation in implementation methods. Here, we discuss the implementations in general, with the understanding that VMMs take advantage of hardware assistance where it is available.

### 18.5.1 The Virtual Machine Life Cycle

Let's begin with the virtual machine life cycle. Whatever the hypervisor type, at the time a virtual machine is created, its creator gives the VMM certain parameters. These parameters usually include the number of CPUs, amount of memory, networking details, and storage details that the VMM will take into account when creating the guest. For example, a user might want to create a new guest with two virtual CPUs, 4 GB of memory, 10 GB of disk space, one network interface that gets its IP address via DHCP, and access to the DVD drive.

The VMM then creates the virtual machine with those parameters. In the case of a type 0 hypervisor, the resources are usually dedicated. In this situation, if there are not two virtual CPUs available and unallocated, the creation request in our example will fail. For other hypervisor types, the resources are dedicated or virtualized, depending on the type. Certainly, an IP address cannot be shared, but the virtual CPUs are usually multiplexed on the physical CPUs as discussed in Section 18.6.1. Similarly, memory management usually involves allocating more memory to guests than actually exists in physical memory. This is more complicated and is described in Section 18.6.2.

Finally, when the virtual machine is no longer needed, it can be deleted. When this happens, the VMM first frees up any used disk space and then removes the configuration associated with the virtual machine, essentially forgetting the virtual machine.

These steps are quite simple compared with building, configuring, running, and removing physical machines. Creating a virtual machine from an existing one can be as easy as clicking the "clone" button and providing a new name and IP address. This ease of creation can lead to **virtual machine sprawl**, which occurs when there are so many virtual machines on a system that their use, history, and state become confusing and difficult to track.

### 18.5.2 Type 0 Hypervisor

Type 0 hypervisors have existed for many years under many names, including "partitions" and "domains." They are a hardware feature, and that brings its own positives and negatives. Operating systems need do nothing special to take advantage of their features. The VMM itself is encoded in the firmware and

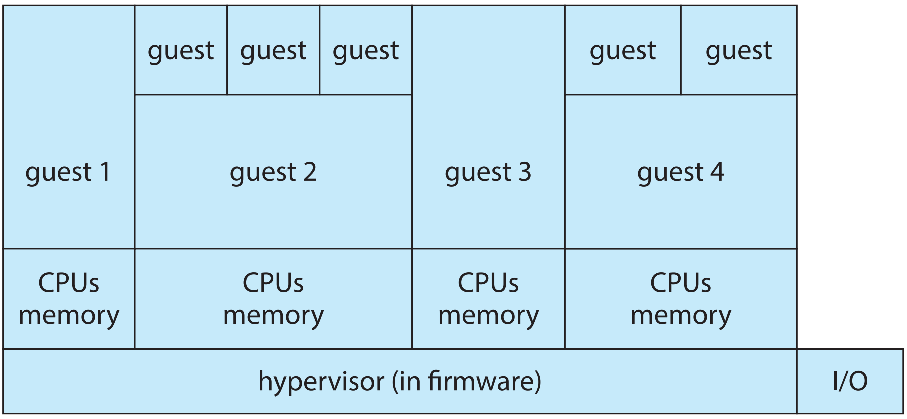


**Figure 18.5** Type 0 hypervisor.

loaded at boot time. In turn, it loads the guest images to run in each partition. The feature set of a type 0 hypervisor tends to be smaller than those of the other types because it is implemented in hardware. For example, a system might be split into four virtual systems, each with dedicated CPUs, memory, and I/O devices. Each guest believes that it has dedicated hardware because it does, simplifying many implementation details.

I/O presents some difficulty, because it is not easy to dedicate I/O devices to guests if there are not enough. What if a system has two Ethernet ports and more than two guests, for example? Either all guests must get their own I/O devices, or the system must provided I/O device sharing. In these cases, the hypervisor manages shared access or grants all devices to a **control partition**. In the control partition, a guest operating system provides services (such as networking) via daemons to other guests, and the hypervisor routes I/O requests appropriately. Some type 0 hypervisors are even more sophisticated and can move physical CPUs and memory between running guests. In these cases, the guests are paravirtualized, aware of the virtualization and assisting in its execution. For example, a guest must watch for signals from the hardware or VMM that a hardware change has occurred, probe its hardware devices to detect the change, and add or subtract CPUs or memory from its available resources.

Because type 0 virtualization is very close to raw hardware execution, it should be considered separately from the other methods discussed here. A type 0 hypervisor can run multiple guest operating systems (one in each hardware partition). All of those guests, because they are running on raw hardware, can in turn be VMMs. Essentially, each guest operating system in a type 0 hypervisor is a native operating system with a subset of hardware made available to it. Because of that, each can have its own guest operating systems (Figure 18.5). Other types of hypervisors usually cannot provide this virtualization-within-virtualization functionality.

### 18.5.3 Type 1 Hypervisor

Type 1 hypervisors are commonly found in company data centers and are, in a sense, becoming "the data-center operating system." They are special-purpose operating systems that run natively on the hardware, but rather than providing system calls and other interfaces for running programs, they create, run, and manage guest operating systems. In addition to running on standard hardware, they can run on type 0 hypervisors, but not on other type 1 hypervisors. Whatever the platform, guests generally do not know they are running on anything but the native hardware.

Type 1 hypervisors run in kernel mode, taking advantage of hardware protection. Where the host CPU allows, they use multiple modes to give guest operating systems their own control and improved performance. They implement device drivers for the hardware they run on, since no other component could do so. Because they are operating systems, they must also provide CPU scheduling, memory management, I/O management, protection, and even security. Frequently, they provide APIs, but those APIs support applications in guests or external applications that supply features like backups, monitoring, and security. Many type 1 hypervisors are closed-source commercial offerings, such as VMware ESX, while some are open source or hybrids of open and closed source, such as Citrix XenServer and its open Xen counterpart.

By using type 1 hypervisors, data-center managers can control and manage the operating systems and applications in new and sophisticated ways. An important benefit is the ability to consolidate more operating systems and applications onto fewer systems. For example, rather than having ten systems running at 10 percent utilization each, a data center might have one server manage the entire load. If utilization increases, guests and their applications can be moved to less-loaded systems live, without interruption of service. Using snapshots and cloning, the system can save the states of guests and duplicate those states—a much easier task than restoring from backups or installing manually or via scripts and tools. The price of this increased manageability is the cost of the VMM (if it is a commercial product), the need to learn new management tools and methods, and the increased complexity.

Another type of type 1 hypervisor includes various general-purpose operating systems with VMM functionality. Here, an operating system such as RedHat Enterprise Linux, Windows, or Oracle Solaris performs its normal duties as well as providing a VMM allowing other operating systems to run as guests. Because of their extra duties, these hypervisors typically provide fewer virtualization features than other type 1 hypervisors. In many ways, they treat a guest operating system as just another process, but they provide special handling when the guest tries to execute special instructions.

### 18.5.4 Type 2 Hypervisor

Type 2 hypervisors are less interesting to us as operating-system explorers, because there is very little operating-system involvement in these application-level virtual machine managers. This type of VMM is simply another process run and managed by the host, and even the host does not know that virtualization is happening within the VMM.

Type 2 hypervisors have limits not associated with some of the other types. For example, a user needs administrative privileges to access many of the hardware assistance features of modern CPUs. If the VMM is being run by a standard user without additional privileges, the VMM cannot take advantage of these features. Due to this limitation, as well as the extra overhead of running

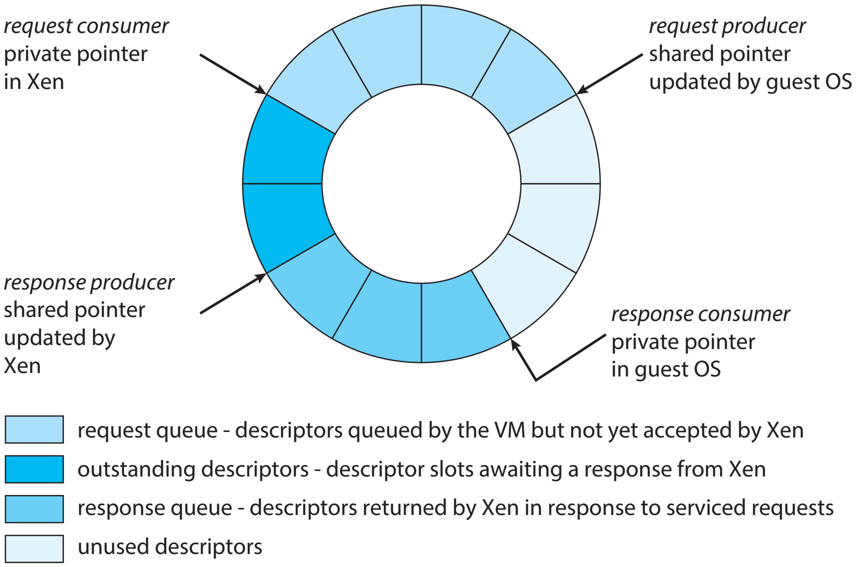

*request consumer* private pointer in Xen
*request producer* shared pointer updated by guest OS
*response producer* shared pointer updated by Xen
*response consumer* private pointer in guest OS
request queue - descriptors queued by the VM but not yet accepted by Xen
outstanding descriptors - descriptor slots awaiting a response from Xen
response queue - descriptors returned by Xen in response to serviced requests
unused descriptors

**Figure 18.6** Xen I/O via shared circular buffer.[1]

a general-purpose operating system as well as guest operating systems, type 2 hypervisors tend to have poorer overall performance than type 0 or type 1.

As is often the case, the limitations of type 2 hypervisors also provide some benefits. They run on a variety of general-purpose operating systems, and running them requires no changes to the host operating system. A student can use a type 2 hypervisor, for example, to test a non-native operating system without replacing the native operating system. In fact, on an Apple laptop, a student could have versions of Windows, Linux, Unix, and less common operating systems all available for learning and experimentation.

### 18.5.5 Paravirtualization

As we've seen, paravirtualization works differently than the other types of virtualization. Rather than try to trick a guest operating system into believing it has a system to itself, paravirtualization presents the guest with a system that is similar but not identical to the guest's preferred system. The guest must be modified to run on the paravirtualized virtual hardware. The gain for this extra work is more efficient use of resources and a smaller virtualization layer.

The Xen VMM became the leader in paravirtulization by implementing several techniques to optimize the performance of guests as well as of the host system. For example, as mentioned earlier, some VMMs present virtual devices to guests that appear to be real devices. Instead of taking that approach, the Xen VMM presented clean and simple device abstractions that allow efficient I/O as well as good I/O-related communication between the guest and the VMM. For each device used by each guest, there was a circular buffer shared by the guest and the VMM via shared memory. Read and write data are placed in this buffer, as shown in Figure 18.6.

---

[1] Barham, Paul. "Xen and the Art of Virtualization". SOSP '03 Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, p 164-177. ©2003 Association for Computing Machinery, Inc

For memory management, Xen did not implement nested page tables. Rather, each guest had its own set of page tables, set to read-only. Xen required the guest to use a specific mechanism, a **hypercall** from the guest to the hypervisor VMM, when a page-table change was needed. This meant that the guest operating system's kernel code must have been changed from the default code to these Xen-specific methods. To optimize performance, Xen allowed the guest to queue up multiple page-table changes asynchronously via hypercalls and then checked to ensure that the changes were complete before continuing operation.

Xen allowed virtualization of x86 CPUs without the use of binary translation, instead requiring modifications in the guest operating systems like the one described above. Over time, Xen has taken advantage of hardware features supporting virtualization. As a result, it no longer requires modified guests and essentially does not need the paravirtualization method. Paravirtualization is still used in other solutions, however, such as type 0 hypervisors.

### 18.5.6 Programming-Environment Virtualization

Another kind of virtualization, based on a different execution model, is the virtualization of programming ***environments.*** Here, a programming language is designed to run within a custom-built virtualized environment. For example, Oracle's Java has many features that depend on its running in the **Java virtual machine** (**JVM**), including specific methods for security and memory management.

If we define virtualization as including only duplication of hardware, this is not really virtualization at all. But we need not limit ourselves to that definition. Instead, we can define a virtual environment, based on APIs, that provides a set of features we want to have available for a particular language and programs written in that language. Java programs run within the JVM environment, and the JVM is compiled to be a native program on systems on which it runs. This arrangement means that Java programs are written once and then can run on any system (including all of the major operating systems) on which a JVM is available. The same can be said of **interpreted languages**, which run inside programs that read each instruction and interpret it into native operations.

### 18.5.7 Emulation

Virtualization is probably the most common method for running applications designed for one operating system on a different operating system, but on the same CPU. This method works relatively efficiently because the applications were compiled for the instruction set that the target system uses.

But what if an application or operating system needs to run on a different CPU? Here, it is necessary to translate all of the source CPU's instructions so that they are turned into the equivalent instructions of the target CPU. Such an environment is no longer virtualized but rather is fully emulated.

**Emulation** is useful when the host system has one system architecture and the guest system was compiled for a different architecture. For example, suppose a company has replaced its outdated computer system with a new system but would like to continue to run certain important programs that were compiled for the old system. The programs could be run in an emulator that translates each of the outdated system's instructions into the native instruction set of the new system. Emulation can increase the life of programs and allow us to explore old architectures without having an actual old machine.

As may be expected, the major challenge of emulation is performance. Instruction-set emulation may run an order of magnitude slower than native instructions, because it may take ten instructions on the new system to read, parse, and simulate an instruction from the old system. Thus, unless the new machine is ten times faster than the old, the program running on the new machine will run more slowly than it did on its native hardware. Another challenge for emulator writers is that it is difficult to create a correct emulator because, in essence, this task involves writing an entire CPU in software.

In spite of these challenges, emulation is very popular, particularly in gaming circles. Many popular video games were written for platforms that are no longer in production. Users who want to run those games frequently can find an emulator of such a platform and then run the game unmodified within the emulator. Modern systems are so much faster than old game consoles that even the Apple iPhone has game emulators and games available to run within them.

### 18.5.8 Application Containment

The goal of virtualization in some instances is to provide a method to segregate applications, manage their performance and resource use, and create an easy way to start, stop, move, and manage them. In such cases, perhaps full-fledged virtualization is not needed. If the applications are all compiled for the same operating system, then we do not need complete virtualization to provide these features. We can instead use application containment.

Consider one example of application containment. Starting with version 10, Oracle Solaris has included **containers**, or **zones**, that create a virtual layer between the operating system and the applications. In this system, only one kernel is installed, and the hardware is not virtualized. Rather, the operating system and its devices are virtualized, providing processes within a zone with the impression that they are the only processes on the system. One or more containers can be created, and each can have its own applications, network stacks, network address and ports, user accounts, and so on. CPU and memory resources can be divided among the zones and the system-wide processes. Each zone, in fact, can run its own scheduler to optimize the performance of its applications on the allotted resources. Figure 18.7 shows a Solaris 10 system with two containers and the standard "global" user space.

Containers are much lighter weight than other virtualization methods. That is, they use fewer system resources and are faster to instantiate and destroy, more similar to processes than virtual machines. For this reason, they are becoming more commonly used, especially in cloud computing. FreeBSD was perhaps the first operating system to include a container-like feature (called "jails"), and AIX has a similar feature. Linux added the **LXC** container feature in 2014. It is now included in the common Linux distributions via

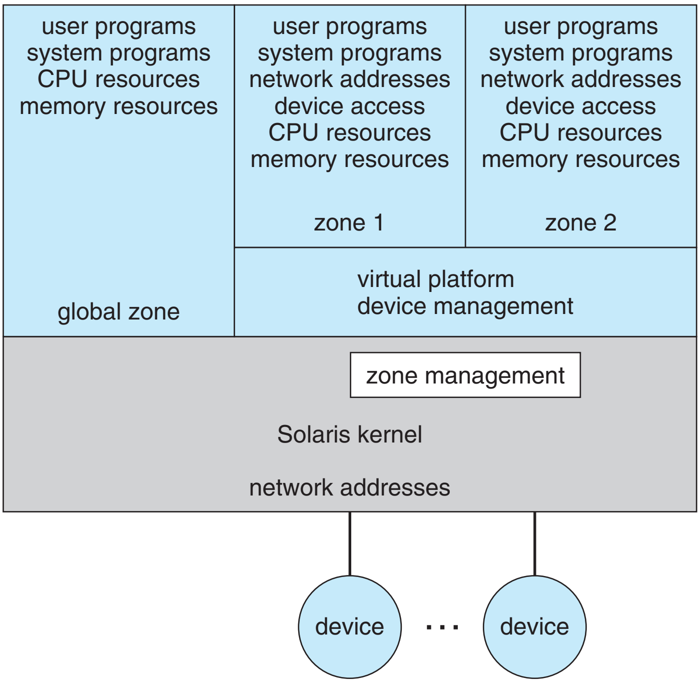

user programs
system programs
CPU resources
memory resources
global zone
user programs
system programs
network addresses
device access
CPU resources
memory resources
zone 1
user programs
system programs
network addresses
device access
CPU resources
memory resources
zone 2
virtual platform
device management
zone management
Solaris kernel
network addresses
device
. . .
device

**Figure 18.7** Solaris 10 with two zones.

a flag in the `clone()` system call. (The source code for LXCis available at https://linuxcontainers.org/lxc/downloads.)

Containers are also easy to automate and manage, leading to orchestration tools like **docker** and **Kubernetes**. Orchestration tools are means of automating and coordinating systems and services. Their aim is to make it simple to run entire suites of distributed applications, just as operating systems make it simple to run a single program. These tools offer rapid deployment of full applications, consisting of many processes within containers, and also offer monitoring and other administration features. For more on docker, see https://www.docker.com/what-docker. Information about Kubernetes can be found at https://kubernetes.io/docs/concepts/overview/what-is-kubernetes.

## 18.6 Virtualization and Operating-System Components

Thus far, we have explored the building blocks of virtualization and the various types of virtualization. In this section, we take a deeper dive into the operating-system aspects of virtualization, including how the VMM provides core operating-system functions like scheduling, I/O, and memory management. Here, we answer questions such as these: How do VMMs schedule CPU use when guest operating systems believe they have dedicated CPUs? How can memory management work when many guests require large amounts of memory?

### 18.6.1 CPU Scheduling

A system with virtualization, even a single-CPU system, frequently acts like a multiprocessor system. The virtualization software presents one or more virtual CPUs to each of the virtual machines running on the system and then schedules the use of the physical CPUs among the virtual machines.

The significant variations among virtualization technologies make it difficult to summarize the effect of virtualization on scheduling. First, let's consider the general case of VMM scheduling. The VMM has a number of physical CPUs available and a number of threads to run on those CPUs. The threads can be VMM threads or guest threads. Guests are configured with a certain number of virtual CPUs at creation time, and that number can be adjusted throughout the life of the VM. When there are enough CPUs to allocate the requested number to each guest, the VMM can treat the CPUs as dedicated and schedule only a given guest's threads on that guest's CPUs. In this situation, the guests act much like native operating systems running on native CPUs.

Of course, in other situations, there may not be enough CPUs to go around. The VMM itself needs some CPU cycles for guest management and I/O management and can steal cycles from the guests by scheduling its threads across all of the system CPUs, but the impact of this action is relatively minor. More difficult is the case of **overcommitment**, in which the guests are configured for more CPUs than exist in the system. Here, a VMM can use standard scheduling algorithms to make progress on each thread but can also add a fairness aspect to those algorithms. For example, if there are six hardware CPUs and twelve guest-allocated CPUs, the VMM can allocate CPU resources proportionally, giving each guest half of the CPU resources it believes it has. The VMM can still present all twelve virtual CPUs to the guests, but in mapping them onto physical CPUs, the VMM can use its scheduler to distribute them appropriately.

Even given a scheduler that provides fairness, any guest operating-system scheduling algorithm that assumes a certain amount of progress in a given amount of time will most likely be negatively affected by virtualization. Consider a time-sharing operating system that tries to allot 100 milliseconds to each time slice to give users a reasonable response time. Within a virtual machine, this operating system receives only what CPU resources the virtualization system gives it. A 100-millisecond time slice may take much more than 100 milliseconds of virtual CPU time. Depending on how busy the system is, the time slice may take a second or more, resulting in very poor response times for users logged into that virtual machine. The effect on a real-time operating system can be even more serious.

The net outcome of such scheduling is that individual virtualized operating systems receive only a portion of the available CPU cycles, even though they believe they are receiving all of the cycles and indeed are scheduling all of the cycles. Commonly, the time-of-day clocks in virtual machines are incorrect because timers take longer to trigger than they would on dedicated CPUs. Virtualization can thus undo the scheduling-algorithm efforts of the operating systems within virtual machines.

To correct for this, the VMM makes an application available for each type of operating system that the system administrator can install into the guests. This application corrects clock drift and can have other functions, such as virtual device management.

### 18.6.2 Memory Management

Efficient memory use in general-purpose operating systems is a major key to performance. In virtualized environments, there are more users of memory (the guests and their applications, as well as the VMM), leading to more pressure on memory use. Further adding to this pressure is the fact that VMMs typically overcommit memory, so that the total memory allocated to guests exceeds the amount that physically exists in the system. The extra need for efficient memory use is not lost on the implementers of VMMs, who take extensive measures to ensure the optimal use of memory.

For example, VMware ESX uses several methods of memory management. Before memory optimization can occur, the VMM must establish how much real memory each guest should use. To do that, the VMM first evaluates each guest's maximum memory size. General-purpose operating systems do not expect the amount of memory in the system to change, so VMMs must maintain the illusion that the guest has that amount of memory. Next, the VMM computes a target real-memory allocation for each guest based on the configured memory for that guest and other factors, such as overcommitment and system load. It then uses the three low-level mechanisms listed below to reclaim memory from the guests

1. Recall that a guest believes it controls memory allocation via its page-table management, whereas in reality the VMM maintains a nested page table that translates the guest page table to the real page table. The VMM can use this extra level of indirection to optimize the guest's use of memory without the guest's knowledge or help. One approach is to provide double paging. Here, the VMM has its own page-replacement algorithms and loads pages into a backing store that the guest believes is physical memory. Of course, the VMM knows less about the guest's memory access patterns than the guest does, so its paging is less efficient, creating performance problems. VMMs do use this method when other methods are not available or are not providing enough free memory. However, it is not the preferred approach.

2. A common solution is for the VMM to install in each guest a pseudo–device driver or kernel module that the VMM controls. (A **pseudo–device driver** uses device-driver interfaces, appearing to the kernel to be a device driver, but does not actually control a device. Rather, it is an easy way to add kernel-mode code without directly modifying the kernel.) This ***balloon memory manager*** communicates with the VMM and is told to allocate or deallocate memory. If told to allocate, it allocates memory and tells the operating system to pin the allocated pages into physical memory. Recall that pinning locks a page into physical memory so that it cannot be moved or paged out. To the guest, these pinned pages appear to decrease the amount of physical memory it has available, creating memory pressure. The guest then may free up other physical memory to be sure it has enough free memory. Meanwhile, the VMM, knowing that the pages pinned by the balloon process will never be used, removes those physical pages from the guest and allocates them to another guest. At the same time, the guest is using its own memory-management and paging algorithms to manage the available memory, which is the most

efficient option. If memory pressure within the entire system decreases, the VMM will tell the balloon process within the guest to unpin and free some or all of the memory, allowing the guest more pages for its use.

3. Another common method for reducing memory pressure is for the VMM to determine if the same page has been loaded more than once. If this is the case, the VMM reduces the number of copies of the page to one and maps the other users of the page to that one copy. VMware, for example, randomly samples guest memory and creates a hash for each page sampled. That hash value is a “thumbprint” of the page. The hash of every page examined is compared with other hashes stored in a hash table. If there is a match, the pages are compared byte by byte to see if they really are identical. If they are, one page is freed, and its logical address is mapped to the other’s physical address. This technique might seem at first to be ineffective, but consider that guests run operating systems. If multiple guests run the same operating system, then only one copy of the active operating-system pages need be in memory. Similarly, multiple guests could be running the same set of applications, again a likely source of memory sharing.

The overall effect of these mechanisms is to enable guests to behave and perform as if they had the full amount of memory requested, although in reality they have less.

### 18.6.3 I/O

In the area of I/O, hypervisors have some leeway and can be less concerned with how they represent the underlying hardware to their guests. Because of the wide variation in I/O devices, operating systems are used to dealing with varying and flexible I/O mechanisms. For example, an operating system’s device-driver mechanism provides a uniform interface to the operating system whatever the I/O device. Device-driver interfaces are designed to allow third-party hardware manufacturers to provide device drivers connecting their devices to the operating system. Usually, device drivers can be dynamically loaded and unloaded. Virtualization takes advantage of this built-in flexibility by providing specific virtualized devices to guest operating systems.

As described in Section 18.5, VMMs vary greatly in how they provide I/O to their guests. I/O devices may be dedicated to guests, for example, or the VMM may have device drivers onto which it maps guest I/O. The VMM may also provide idealized device drivers to guests. In this case, the guest sees an easy-to-control device, but in reality that simple device driver communicates to the VMM, which sends the requests to a more complicated real device through a more complex real device driver. I/O in virtual environments is complicated and requires careful VMM design and implementation.

Consider the case of a hypervisor and hardware combination that allows devices to be dedicated to a guest and allows the guest to access those devices directly. Of course, a device dedicated to one guest is not available to any other guests, but this direct access can still be useful in some circumstances. The reason to allow direct access is to improve I/O performance. The less the hypervisor has to do to enable I/O for its guests, the faster the I/O can occur. With type 0 hypervisors that provide direct device access, guests can often run at the same speed as native operating systems. When type 0 hypervisors instead provide shared devices, performance may suffer.

With direct device access in type 1 and 2 hypervisors, performance can be similar to that of native operating systems if certain hardware support is present. The hardware needs to provide DMA pass-through with facilities like VT-d, as well as direct interrupt delivery (interrupts going directly to the guests). Given how frequently interrupts occur, it should be no surprise that the guests on hardware without these features have worse performance than if they were running natively.

In addition to direct access, VMMs provide shared access to devices. Consider a disk drive to which multiple guests have access. The VMM must provide protection while the device is being shared, assuring that a guest can access only the blocks specified in the guest's configuration. In such instances, the VMM must be part of every I/O, checking it for correctness as well as routing the data to and from the appropriate devices and guests.

In the area of networking, VMMs also have work to do. General-purpose operating systems typically have one Internet protocol (IP) address, although they sometimes have more than one—for example, to connect to a management network, backup network, and production network. With virtualization, each guest needs at least one IP address, because that is the guest's main mode of communication. Therefore, a server running a VMM may have dozens of addresses, and the VMM acts as a virtual switch to route the network packets to the addressed guests.

The guests can be "directly" connected to the network by an IP address that is seen by the broader network (this is known as **bridging**). Alternatively, the VMM can provide a **network address translation** (**NAT**) address. The NAT address is local to the server on which the guest is running, and the VMM provides routing between the broader network and the guest. The VMM also provides firewalling to guard connections between guests within the system and between guests and external systems.

### 18.6.4 Storage Management

An important question in determining how virtualization works is this: If multiple operating systems have been installed, what and where is the boot disk? Clearly, virtualized environments need to approach storage management differently than do native operating systems. Even the standard multiboot method of slicing the boot disk into partitions, installing a boot manager in one partition, and installing each other operating system in another partition is not sufficient, because partitioning has limits that would prevent it from working for tens or hundreds of virtual machines.

Once again, the solution to this problem depends on the type of hypervisor. Type 0 hypervisors often allow root disk partitioning, partly because these systems tend to run fewer guests than other systems. Alternatively, a disk manager may be part of the control partition, and that disk manager may provide disk space (including boot disks) to the other partitions.

Type 1 hypervisors store the guest root disk (and configuration information) in one or more files in the file systems provided by the VMM. Type 2 hypervisors store the same information in the host operating system's file systems. In essence, a **disk image**, containing all of the contents of the root disk of the guest, is contained in one file in the VMM. Aside from the potential performance problems that causes, this is a clever solution, because it simplifies copying and moving guests. If the administrator wants a duplicate of the guest (for testing, for example), she simply copies the associated disk image of the guest and tells the VMM about the new copy. Booting the new virtual machine brings up an identical guest. Moving a virtual machine from one system to another that runs the same VMM is as simple as halting the guest, copying the image to the other system, and starting the guest there.

Guests sometimes need more disk space than is available in their root disk image. For example, a nonvirtualized database server might use several file systems spread across many disks to store various parts of the database. Virtualizing such a database usually involves creating several files and having the VMM present those to the guest as disks. The guest then executes as usual, with the VMM translating the disk I/O requests coming from the guest into file I/O commands to the correct files.

Frequently, VMMs provide a mechanism to capture a physical system as it is currently configured and convert it to a guest that the VMM can manage and run. This **physical-to-virtual** (**P-to-V**) conversion reads the disk blocks of the physical system's disks and stores them in files on the VMM's system or on shared storage that the VMM can access. VMMs also provide a **virtual-to-physical** (**V-to-P**) procedure for converting a guest to a physical system. This procedure is sometimes needed for debugging: a problem could be caused by the VMM or associated components, and the administrator could attempt to solve the problem by removing virtualization from the problem variables. V-to-P conversion can take the files containing all of the guest data and generate disk blocks on a physical disk, recreating the guest as a native operating system and applications. Once the testing is concluded, the original system can be reused for other purposes when the virtual machine returns to service, or the virtual machine can be deleted and the original system can continue to run.

### 18.6.5 Live Migration

One feature not found in general-purpose operating systems but found in type 0 and type 1 hypervisors is the live migration of a running guest from one system to another. We mentioned this capability earlier. Here, we explore the details of how live migration works and why VMMs can implement it relatively easily while general-purpose operating systems, in spite of some research attempts, cannot.

First, let's consider how live migration works. A running guest on one system is copied to another system running the same VMM. The copy occurs with so little interruption of service that users logged in to the guest, as well as network connections to the guest, continue without noticeable impact. This rather astonishing ability is very powerful in resource management and hardware administration. After all, compare it with the steps necessary without virtualization: we must warn users, shut down the processes, possibly move the binaries, and restart the processes on the new system. Only then can users access the services again. With live migration, we can decrease the load on an overloaded system or make hardware or system changes with no discernable disruption for users.

Live migration is made possible by the well-defined interface between each guest and the VMM and the limited state the VMM maintains for the guest. The VMM migrates a guest via the following steps:

1. The source VMM establishes a connection with the target VMM and confirms that it is allowed to send a guest.
2. The target creates a new guest by creating a new VCPU, new nested page table, and other state storage.
3. The source sends all read-only memory pages to the target.
4. The source sends all read–write pages to the target, marking them as clean.
5. The source repeats step 4, because during that step some pages were probably modified by the guest and are now dirty. These pages need to be sent again and marked again as clean.
6. When the cycle of steps 4 and 5 becomes very short, the source VMM freezes the guest, sends the VCPU's final state, other state details, and the final dirty pages, and tells the target to start running the guest. Once the target acknowledges that the guest is running, the source terminates the guest.

This sequence is shown in Figure 18.8.

We conclude this discussion with a few interesting details and limitations concerning live migration. First, for network connections to continue uninterrupted, the network infrastructure needs to understand that a MAC address—the hardware networking address—can move between systems. Before virtualization, this did not happen, as the MAC address was tied to physical hardware. With virtualization, the MAC must be movable for existing networking connections to continue without resetting. Modern network switches understand this and route traffic wherever the MAC address is, even accommodating a move.

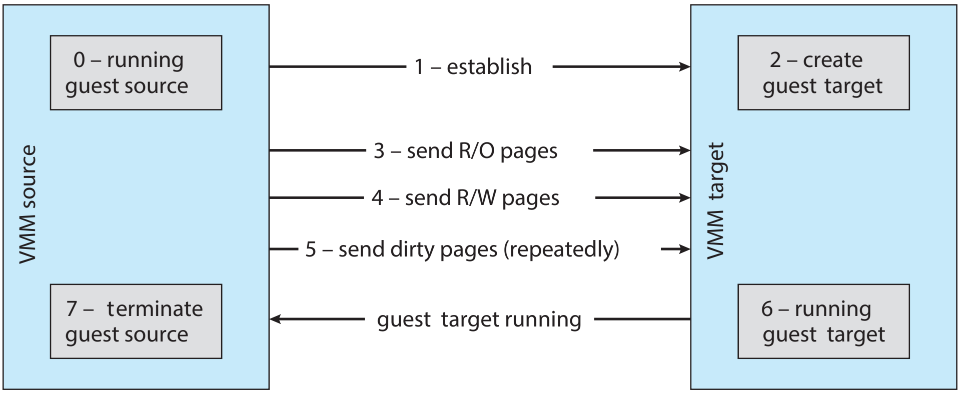

VMM source
0 – running guest source
7 – terminate guest source
1 – establish
3 – send R/O pages
4 – send R/W pages
5 – send dirty pages (repeatedly)
guest target running
VMM target
2 – create guest target
6 – running guest target

**Figure 18.8** Live migration of a guest between two servers.

A limitation of live migration is that no disk state is transferred. One reason live migration is possible is that most of the guest's state is maintained within the guest—for example, open file tables, system-call state, kernel state, and so on. Because disk I/O is much slower than memory access, however, and used disk space is usually much larger than used memory, disks associated with the guest cannot be moved as part of a live migration. Rather, the disk must be remote to the guest, accessed over the network. In that case, disk access state is maintained within the guest, and network connections are all that matter to the VMM. The network connections are maintained during the migration, so remote disk access continues. Typically, NFS, CIFS, or iSCSI is used to store virtual machine images and any other storage a guest needs access to. These network-based storage accesses simply continue when the network connections are continued once the guest has been migrated.

Live migration makes it possible to manage data centers in entirely new ways. For example, virtualization management tools can monitor all the VMMs in an environment and automatically balance resource use by moving guests between the VMMs. These tools can also optimize the use of electricity and cooling by migrating all guests off selected servers if other servers can handle the load and powering down the selected servers entirely. If the load increases, the tools can power up the servers and migrate guests back to them.

## 18.7 Examples

Despite the advantages of virtual machines, they received little attention for a number of years after they were first developed. Today, however, virtual machines are coming into greater use as a means of solving system compatibility problems. In this section, we explore two popular contemporary virtual machines: the VMware Workstation and the Java virtual machine. These virtual machines can typically run on top of operating systems of any of the design types discussed in earlier chapters.

### 18.7.1 VMware

**VMware Workstation** is a popular commercial application that abstracts Intel x86 and compatible hardware into isolated virtual machines. VMware Workstation is a prime example of a Type 2 hypervisor. It runs as an application on a host operating system such as Windows or Linux and allows this host system to run several different guest operating systems concurrently as independent virtual machines.

The architecture of such a system is shown in Figure 18.9. In this scenario, Linux is running as the host operating system, and FreeBSD, Windows NT, and Windows XP are running as guest operating systems. At the heart of VMware is the virtualization layer, which abstracts the physical hardware into isolated virtual machines running as guest operating systems. Each virtual machine has its own virtual CPU, memory, disk drives, network interfaces, and so forth.

The physical disk that the guest owns and manages is really just a file within the file system of the host operating system. To create an identical guest, we can simply copy the file. Copying the file to another location protects the guest against a disaster at the original site. Moving the file to another location

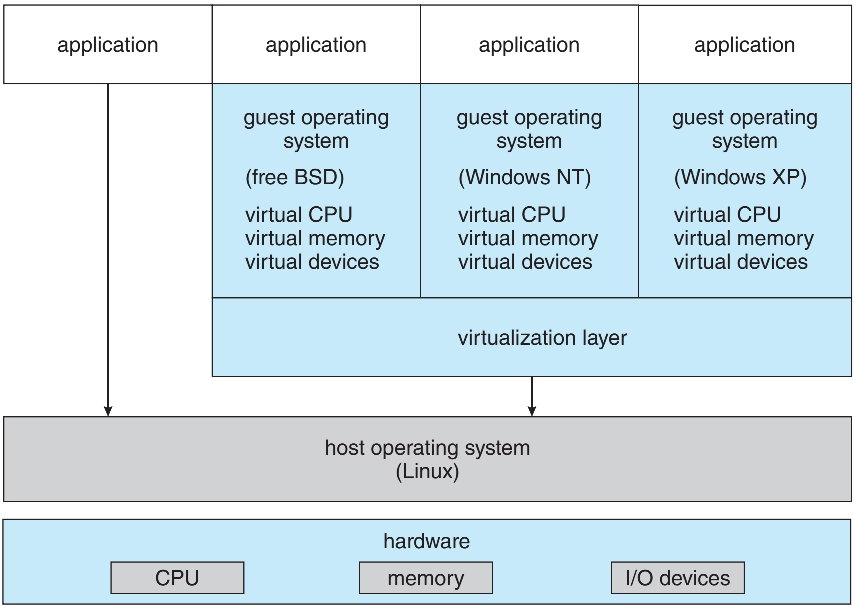

application
application
application
application
guest operating system
(free BSD)
virtual CPU
virtual memory
virtual devices
guest operating system
(Windows NT)
virtual CPU
virtual memory
virtual devices
guest operating system
(Windows XP)
virtual CPU
virtual memory
virtual devices
virtualization layer
host operating system
(Linux)
hardware
CPU
memory
I/O devices

**Figure 18.9** VMware Workstation architecture.

moves the guest system. Such capabilities, as explained earlier, can improve the efficiency of system administration as well as system resource use.

### 18.7.2 The Java Virtual Machine

Java is a popular object-oriented programming language introduced by Sun Microsystems in 1995. In addition to a language specification and a large API library, Java provides a specification for a Java virtual machine, or JVM. Java therefore is an example of programming-environment virtualization, as discussed in Section 18.5.6.

Java objects are specified with the `class` construct; a Java program consists of one or more classes. For each Java class, the compiler produces an architecture-neutral **bytecode** output (`.class`) file that will run on any implementation of the JVM.

The JVM is a specification for an abstract computer. It consists of a **class loader** and a Java interpreter that executes the architecture-neutral bytecodes, as diagrammed in Figure 18.10. The class loader loads the compiled `.class` files from both the Java program and the Java API for execution by the Java interpreter. After a class is loaded, the verifier checks that the `.class` file is valid Java bytecode and that it does not overflow or underflow the stack. It also ensures that the bytecode does not perform pointer arithmetic, which could provide illegal memory access. If the class passes verification, it is run by the Java interpreter. The JVM also automatically manages memory by performing **garbage collection**—the practice of reclaiming memory from objects no longer in use and returning it to the system. Much research focuses on garbage collection algorithms for increasing the performance of Java programs in the virtual machine.

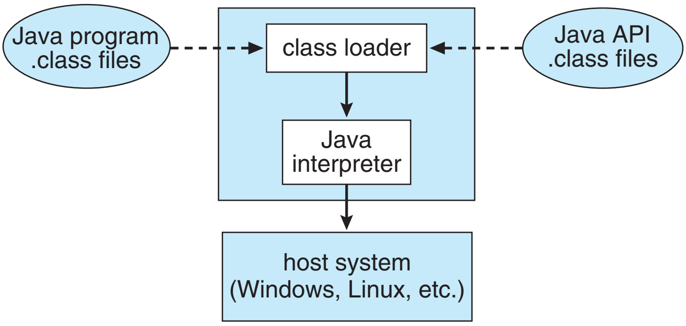


**Figure 18.10** The Java virtual machine.

The JVM may be implemented in software on top of a host operating system, such as Windows, Linux, or macOS, or as part of a web browser. Alternatively, the JVM may be implemented in hardware on a chip specifically designed to run Java programs. If the JVM is implemented in software, the Java interpreter interprets the bytecode operations one at a time. A faster software technique is to use a **just-in-time** (**JIT**) compiler. Here, the first time a Java method is invoked, the bytecodes for the method are turned into native machine language for the host system. These operations are then cached so that subsequent invocations of a method are performed using the native machine instructions, and the bytecode operations need not be interpreted all over again. Running the JVM in hardware is potentially even faster. Here, a special Java chip executes the Java bytecode operations as native code, thus bypassing the need for either a software interpreter or a just-in-time compiler.

## 18.8 Virtualization Research

As mentioned earlier, machine virtualization has enjoyed growing popularity in recent years as a means of solving system compatibility problems. Research has expanded to cover many other uses of machine virtualization, including support for microservices running on library operating systems and secure partitioning of resources in embedded systems. Consequently, quite a lot of interesting, active research is underway.

Frequently, in the context of cloud computing, the same application is run on thousands of systems. To better manage those deployments, they can be virtualized. But consider the execution stack in that case—the application on top of a service-rich general-purpose operating system within a virtual machine managed by a hypervisor. Projects like **unikernels**, built on **library operating systems**, aim to improve efficiency and security in these environments. Unikernels are specialized machine images, using one address space, that shrink the attack surface and resource footprint of deployed applications. In essence, they compile the application, the system libraries it calls, and the kernel services it uses into a single binary that runs within a virtual environment (or even on bare metal). While research into changing how operating system kernels, hardware, and applications interact is not new (see https://pdos.csail.mit.edu/6.828/2005/readings/engler95exokernel.pdf, for example), cloud computing and virtualization have created renewed interest in the area. See http://unikernel.org for more details.

The virtualization instructions in modern CPUs have given rise to a new branch of virtualization research focusing not on more efficient use of hardware but rather on better control of processes. Partitioning hypervisors partition the existing machine physical resources amongst guests, thereby fully committing rather than overcommitting machine resources. Partitioning hypervisors can securely extend the features of an existing operating system via functionality in another operating system (run in a separate guest VM domain), running on a subset of machine physical resources. This avoids the tedium of writing an entire operating system from scratch. For example, a Linux system that lacks real-time capabilities for safety- and security-critical tasks can be extended with a lightweight real-time operating system running in its own virtual machine. Traditional hypervisors have higher overhead than running native tasks, so a new type of hypervisor is needed.

Each task runs within a virtual machine, but the hypervisor only initializes the system and starts the tasks and is not involved with continuing operation. Each virtual machine has its own allocated hardware and is free to manage that hardware without interference from the hypervisor. Because the hypervisor does not interrupt task operations and is not called by the tasks, the tasks can have real-time aspects and can be much more secure.

Within the class of partitioning hypervisors are the **Quest-V**, **eVM**, **Xtratum** and **Siemens Jailhouse** projects. These are **separation hypervisors** (see http://www.csl.sri.com/users/rushby/papers/sosp81.pdf) that use virtualization to partition separate system components into a chip-level distributed system. Secure shared memory channels are then implemented using hardware extended page tables so that separate sandboxed guests can communicate with one another. The targets of these projects are areas such as robotics, self-driving cars, and the Internet of Things. See https://www.cs.bu.edu/richwest/papers/west-tocs16.pdf for more details.

## 18.9 Summary

- Virtualization is a method for providing a guest with a duplicate of a system's underlying hardware. Multiple guests can run on a given system, each believing that it is the native operating system and is in full control.

- Virtualization started as a method to allow IBM to segregate users and provide them with their own execution environments on IBM mainframes. Since then, thanks to improvements in system and CPU performance and innovative software techniques, virtualization has become a common feature in data centers and even on personal computers. Because of its popularity, CPU designers have added features to support virtualization. This snowball effect is likely to continue, with virtualization and its hardware support increasing over time.

- The virtual machine manager, or hypervisor, creates and runs the virtual machine. Type 0 hypervisors are implemented in the hardware and require modifications to the operating system to ensure proper operation. Some type 0 hypervisors offer an example of paravirtualization, in which the operating system is aware of virtualization and assists in its execution.

- Type 1 hypervisors provide the environment and features needed to create, run, and manage guest virtual machines. Each guest includes all of the software typically associated with a full native system, including the operating system, device drivers, applications, user accounts, and so on.
- Type 2 hypervisors are simply applications that run on other operating systems, which do not know that virtualization is taking place. These hypervisors do not have hardware or host support so must perform all virtualization activities in the context of a process.
- Programming-environment virtualization is part of the design of a programming language. The language specifies a containing application in which programs run, and this application provides services to the programs.
- Emulation is used when a host system has one architecture and the guest was compiled for a different architecture. Every instruction the guest wants to execute must be translated from its instruction set to that of the native hardware. Although this method involves some performance penalty, it is balanced by the usefulness of being able to run old programs on newer, incompatible hardware or run games designed for old consoles on modern hardware.
- Implementing virtualization is challenging, especially when hardware support is minimal. The more features provided by the system, the easier virtualization is to implement and the better the performance of the guests.
- VMMs take advantage of whatever hardware support is available when optimizing CPU scheduling, memory management, and I/O modules to provide guests with optimum resource use while protecting the VMM from the guests and the guests from one another.
- Current research is extending the uses of virtualization. Unikernels aim to increase efficiency and decrease security attack surface by compiling an application, its libraries, and the kernel resources the application needs into one binary with one address space that runs within a virtual machine. Partitioning hypervisors provide secure execution, real-time operation, and other features traditionally only available to applications running on dedicated hardware.

## Further Reading

The original IBM virtual machine is described in [Meyer and Seawright (1970)]. [Popek and Goldberg (1974)] established the characteristics that help define VMMs. Methods of implementing virtual machines are discussed in [Agesen et al. (2010)].

Intel x86 hardware virtualization support is described in [Neiger et al. (2006)]. AMD hardware virtualization support is described in a white paper available at http://developer.amd.com/assets/NPT-WP-1%201-final-TM.pdf.

Memory management in VMware is described in [Waldspurger (2002)]. [Gordon et al. (2012)] propose a solution to the problem of I/O overhead in virtualized environments. Some protection challenges and attacks in virtual environments are discussed in [Wojtczuk and Ruthkowska (2011)].

For early work on alternative kernel designs, see https://pdos.csail.mit.edu/6.828/2005/readings/engler95exokernel.pdf. For more on unikernels, see [West et al. (2016)] and http://unikernel.org. Partitioning hypervisors are discussed in http://ethdocs.org/en/latest/introduction/what-is-ethereum.html, and https://lwn.net/Articles/578295 and [Madhavapeddy et al. (2013)]. Quest-V, a separation hypervisor, is detailed in http://www.csl.sri.com/users/rushby/papers/sosp81.pdf and https://www.cs.bu.edu/ richwest/papers/west-tocs16.pdf.

The open-source ***VirtualBox*** project is available from http://www.virtualbox.org. The source code for LXC is available at https://linuxcontainers.org/lxc/downloads.

For more on docker, see https://www.docker.com/what-docker. Information about Kubernetes can be found at https://kubernetes.io/docs/concepts/overview/what-is-kubernetes.

# Bibliography

**[Agesen et al. (2010)]** O. Agesen, A. Garthwaite, J. Sheldon, and P. Subrahmanyam, “The Evolution of an x86 Virtual Machine Monitor”, *Proceedings of the ACM Symposium on Operating Systems Principles* (2010), pages 3–18.

**[Gordon et al. (2012)]** A. Gordon, N. A. N. Har’El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir, “ELI: Bare-metal Performance for I/O Virtualization”, *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (2012), pages 411–422.

**[Madhavapeddy et al. (2013)]** A. Madhavapeddy, R. Mirtier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, “Unikernels: Library Operating Systems for the Cloud” (2013).

**[Meyer and Seawright (1970)]** R. A. Meyer and L. H. Seawright, “A Virtual Machine Time-Sharing System”, *IBM Systems Journal*, Volume 9, Number 3 (1970), pages 199–218.

**[Neiger et al. (2006)]** G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig, “Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization”, *Intel Technology Journal*, Volume 10, (2006).

**[Popek and Goldberg (1974)]** G. J. Popek and R. P. Goldberg, “Formal Requirements for Virtualizable Third Generation Architectures”, *Communications of the ACM*, Volume 17, Number 7 (1974), pages 412–421.

**[Waldspurger (2002)]** C. Waldspurger, “Memory Resource Management in VMware ESX Server”, *Operating Systems Review*, Volume 36, Number 4 (2002), pages 181–194.

**[West et al. (2016)]** R. West, Y. Li, E. Missimer, and M. Danish, “A Virtualized Separation Kernel for Mixed Criticality Systems”, Volume 34, (2016).

**[Wojtczuk and Ruthkowska (2011)]** R. Wojtczuk and J. Ruthkowska, “Following the White Rabbit: Software Attacks Against Intel VT-d Technology”, *The Invisible Things Lab’s blog* (2011).

## Chapter 18 Exercises

**18.1** Describe the three types of traditional hypervisors.

**18.2** Describe four virtualization-like execution environments, and explain how they differ from “true” virtualization.

**18.3** Describe four benefits of virtualization.

**18.4** Why are VMMs unable to implement trap-and-emulate-based virtualization on some CPUs? Lacking the ability to trap and emulate, what method can a VMM use to implement virtualization?

**18.5** What hardware assistance for virtualization can be provided by modern CPUs?

**18.6** Why is live migration possible in virtual environments but much less possible for a native operating system?

# Networks and Distributed Systems

CHAPTER 19

**Updated by Sarah Diesburg**

A distributed system is a collection of processors that do not share memory or a clock. Instead, each node has its own local memory. The nodes communicate with one another through various networks, such as high-speed buses. Distributed systems are more relevant than ever, and you have almost certainly used some sort of distributed service. Applications of distributed systems range from providing transparent access to files inside an organization, to large-scale cloud file and photo storage services, to business analysis of trends on large data sets, to parallel processing of scientific data, and more. In fact, the most basic example of a distributed system is one we are all likely very familiar with—the Internet.

In this chapter, we discuss the general structure of distributed systems and the networks that interconnect them. We also contrast the main differences in the types and roles of current distributed system designs. Finally, we investigate some of the basic designs and design challenges of distributed file systems.

## CHAPTER OBJECTIVES

- Explain the advantages of networked and distributed systems.
- Provide a high-level overview of the networks that interconnect distributed systems.
- Define the roles and types of distributed systems in use today.
- Discuss issues concerning the design of distributed file systems.

## 19.1 Advantages of Distributed Systems

A **distributed system** is a collection of loosely coupled nodes interconnected by a communication network. From the point of view of a specific node in a distributed system, the rest of the nodes and their respective resources are remote, whereas its own resources are local.

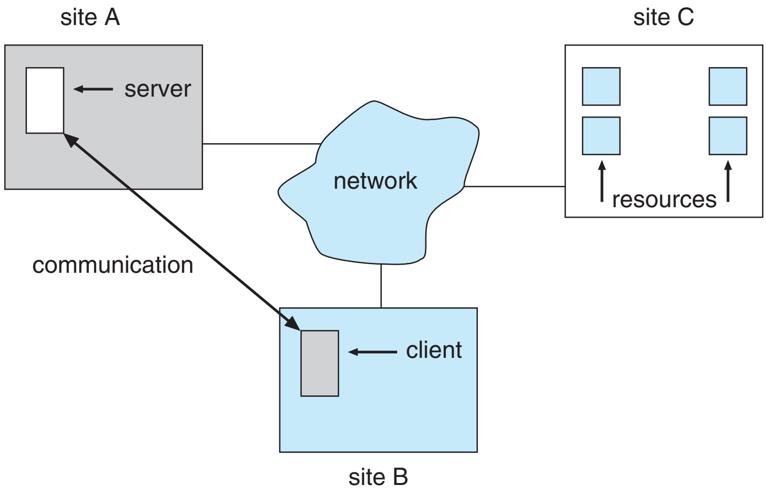


**Figure 19.1** A client-server distributed system.

The nodes in a distributed system may vary in size and function. They may include small microprocessors, personal computers, and large general-purpose computer systems. These processors are referred to by a number of names, such as *processors*, *sites*, *machines*, and *hosts*, depending on the context in which they are mentioned. We mainly use *site* to indicate the location of a machine and *node* to refer to a specific system at a site. Nodes can exist in a *client–server* configuration, a *peer-to-peer* configuration, or a hybrid of these. In the common client-server configuration, one node at one site, the *server*, has a resource that another node, the *client* (or user), would like to use. A general structure of a client–server distributed system is shown in Figure 19.1. In a peer-to-peer configuration, there are no servers or clients. Instead, the nodes share equal responsibilities and can act as both clients and servers.

When several sites are connected to one another by a communication network, users at the various sites have the opportunity to exchange information. At a low level, **messages** are passed between systems, much as messages are passed between processes in the single-computer message system discussed in Section 3.4. Given message passing, all the higher-level functionality found in standalone systems can be expanded to encompass the distributed system. Such functions include file storage, execution of applications, and remote procedure calls (RPCs).

There are three major reasons for building distributed systems: resource sharing, computational speedup, and reliability. In this section, we briefly discuss each of them.

### 19.1.1 Resource Sharing

If a number of different sites (with different capabilities) are connected to one another, then a user at one site may be able to use the resources available at another. For example, a user at site A may query a database located at site B. Meanwhile, a user at site B may access a file that resides at site A. In general, **resource sharing** in a distributed system provides mechanisms for sharing files at remote sites, processing information in a distributed database, printing files at remote sites, using remote specialized hardware devices such as a supercomputer or a **graphics processing unit** (**GPU**), and performing other operations.

### 19.1.2 Computation Speedup

If a particular computation can be partitioned into subcomputations that can run concurrently, then a distributed system allows us to distribute the subcomputations among the various sites. The subcomputations can be run concurrently and thus provide **computation speedup**. This is especially relevant when doing large-scale processing of big data sets (such as analyzing large amounts of customer data for trends). In addition, if a particular site is currently overloaded with requests, some of them can be moved or rerouted to other, more lightly loaded sites. This movement of jobs is called **load balancing** and is common among distributed system nodes and other services provided on the Internet.

### 19.1.3 Reliability

If one site fails in a distributed system, the remaining sites can continue operating, giving the system better reliability. If the system is composed of multiple large autonomous installations (that is, general-purpose computers), the failure of one of them should not affect the rest. If, however, the system is composed of diversified machines, each of which is responsible for some crucial system function (such as the web server or the file system), then a single failure may halt the operation of the whole system. In general, with enough redundancy (in both hardware and data), the system can continue operation even if some of its nodes have failed.

The failure of a node or site must be detected by the system, and appropriate action may be needed to recover from the failure. The system must no longer use the services of that site. In addition, if the function of the failed site can be taken over by another site, the system must ensure that the transfer of function occurs correctly. Finally, when the failed site recovers or is repaired, mechanisms must be available to integrate it back into the system smoothly.

## 19.2 Network Structure

To completely understand the roles and types of distributed systems in use today, we need to understand the networks that interconnect them. This section serves as a network primer to introduce basic networking concepts and challenges as they relate to distributed systems. The rest of the chapter specifically discusses distributed systems.

There are basically two types of networks: **local-area networks** (**LAN**) and **wide-area networks** (**WAN**). The main difference between the two is the way in which they are geographically distributed. Local-area networks are composed of hosts distributed over small areas (such as a single building or a number of adjacent buildings), whereas wide-area networks are composed of systems distributed over a large area (such as the United States). These differences

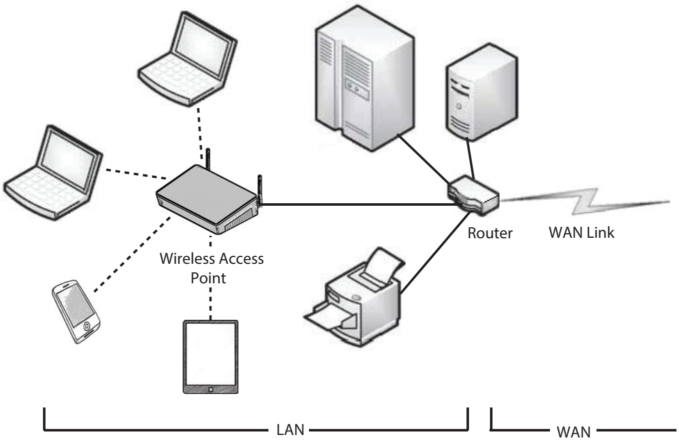


**Figure 19.2** Local-area network.

imply major variations in the speed and reliability of the communications networks, and they are reflected in the distributed system design.

### 19.2.1 Local-Area Networks

Local-area networks emerged in the early 1970s as a substitute for large mainframe computer systems. For many enterprises, it is more economical to have a number of small computers, each with its own self-contained applications, than to have a single large system. Because each small computer is likely to need a full complement of peripheral devices (such as disks and printers), and because some form of data sharing is likely to occur in a single enterprise, it was a natural step to connect these small systems into a network.

LANs, as mentioned, are usually designed to cover a small geographical area, and they are generally used in an office or home environment. All the sites in such systems are close to one another, so the communication links tend to have a higher speed and lower error rate than their counterparts in wide-area networks.

A typical LAN may consist of a number of different computers (including workstations, servers, laptops, tablets, and smartphones), various shared peripheral devices (such as printers and storage arrays), and one or more **routers** (specialized network communication processors) that provide access to other networks (Figure 19.2). Ethernet and **WiFi** are commonly used to construct LANs. *Wireless access points* connect devices to the LAN wirelessly, and they may or may not be routers themselves.

Ethernet networks are generally found in businesses and organizations in which computers and peripherals tend to be nonmobile. These networks use *coaxial, twisted pair,* and/or *fiber optic* cables to send signals. An Ethernet network has no central controller, because it is a multiaccess bus, so new hosts can be added easily to the network. The Ethernet protocol is defined by the IEEE 802.3 standard. Typical Ethernet speeds using common twisted-pair cabling can vary from 10 Mbps to over 10 Gbps, with other types of cabling reaching speeds of 100 Gbps.

WiFi is now ubiquitous and either supplements traditional Ethernet networks or exists by itself. Specifically, WiFi allows us to construct a network without using physical cables. Each host has a wireless transmitter and receiver that it uses to participate in the network. WiFi is defined by the IEEE 802.11 standard. Wireless networks are popular in homes and businesses, as well as public areas such as libraries, Internet cafes, sports arenas, and even buses and airplanes. WiFi speeds can vary from 11 Mbps to over 400 Mbps.

Both the IEEE 802.3 and 802.11 standards are constantly evolving. For the latest information about various standards and speeds, see the references at the end of the chapter.

### 19.2.2 Wide-Area Networks

Wide-area networks emerged in the late 1960s, mainly as an academic research project to provide efficient communication among sites, allowing hardware and software to be shared conveniently and economically by a wide community of users. The first WAN to be designed and developed was the ARPANET. Begun in 1968, the ARPANET has grown from a four-site experimental network to a worldwide network of networks, the **Internet** (also known as the **World Wide Web**), comprising millions of computer systems.

Sites in a WAN are physically distributed over a large geographical area. Typical links are telephone lines, leased (dedicated data) lines, optical cable, microwave links, radio waves, and satellite channels. These communication links are controlled by routers (Figure 19.3) that are responsible for directing traffic to other routers and networks and transferring information among the various sites.

For example, the Internet WAN enables hosts at geographically separate sites to communicate with one another. The host computers typically differ from one another in speed, CPU type, operating system, and so on. Hosts are

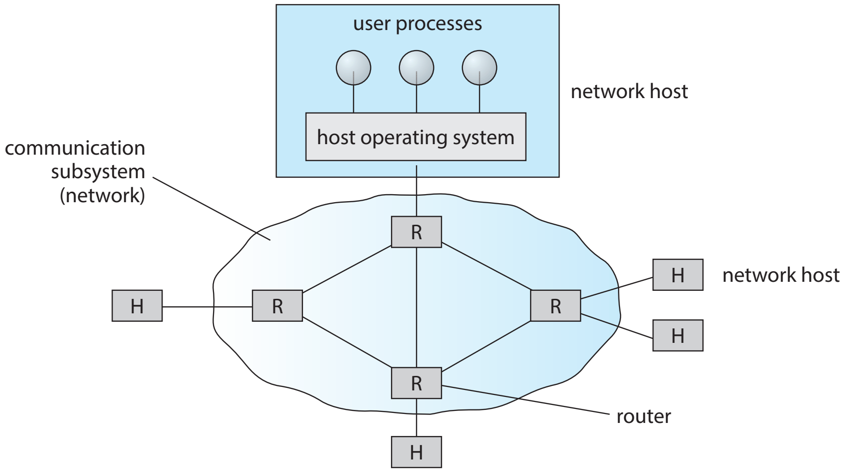


Figure 19.3 Communication processors in a wide-area network.

generally on LANs, which are, in turn, connected to the Internet via regional networks. The regional networks are interlinked with routers to form the worldwide network. Residences can connect to the Internet by either telephone, cable, or specialized Internet service providers that install routers to connect the residences to central services. Of course, there are other WANs besides the Internet. A company, for example, might create its own private WAN for increased security, performance, or reliability.

WANs are generally slower than LANs, although backbone WAN connections that link major cities may have very fast transfer rates through fiber optic cables. In fact, many backbone providers have fiber optic speeds of 40 Gbps or 100 Gbps. (It is generally the links from local **Internet Service Providers** (**ISPs**) to homes or businesses that slow things down.) However, WAN links are being constantly updated to faster technologies as the demand for more speed continues to grow.

Frequently, WANs and LANs interconnect, and it is difficult to tell where one ends and the other starts. Consider the cellular phone data network. Cell phones are used for both voice and data communications. Cell phones in a given area connect via radio waves to a cell tower that contains receivers and transmitters. This part of the network is similar to a LAN except that the cell phones do not communicate with each other (unless two people talking or exchanging data happen to be connected to the same tower). Rather, the towers are connected to other towers and to hubs that connect the tower communications to land lines or other communication media and route the packets toward their destinations. This part of the network is more WAN-like. Once the appropriate tower receives the packets, it uses its transmitters to send them to the correct recipient.

## 19.3 Communication Structure

Now that we have discussed the physical aspects of networking, we turn to the internal workings.

### 19.3.1 Naming and Name Resolution

The first issue in network communication involves the naming of the systems in the network. For a process at site A to exchange information with a process at site B, each must be able to specify the other. Within a computer system, each process has a process identifier, and messages may be addressed with the process identifier. Because networked systems share no memory, however, a host within the system initially has no knowledge about the processes on other hosts.

To solve this problem, processes on remote systems are generally identified by the pair <host name, identifier>, where **host name** is a name unique within the network and **identifie** is a process identifier or other unique number within that host. A host name is usually an alphanumeric identifier, rather than a number, to make it easier for users to specify. For instance, site A might have hosts named *program*, *student*, *faculty*, and *cs*. The host name *program* is certainly easier to remember than the numeric host address *128.148.31.100*.

Names are convenient for humans to use, but computers prefer numbers for speed and simplicity. For this reason, there must be a mechanism to **resolve** the host name into a **host-id** that describes the destination system to the networking hardware. This mechanism is similar to the name-to-address binding that occurs during program compilation, linking, loading, and execution (Chapter 9). In the case of host names, two possibilities exist. First, every host may have a data file containing the names and numeric addresses of all the other hosts reachable on the network (similar to binding at compile time). The problem with this model is that adding or removing a host from the network requires updating the data files on all the hosts. In fact, in the early days of the ARPANET there was a canonical host file that was copied to every system periodically. As the network grew, however, this method became untenable.

The alternative is to distribute the information among systems on the network. The network must then use a protocol to distribute and retrieve the information. This scheme is like execution-time binding. The Internet uses a **domain-name system** (**DNS**) for host-name resolution.

DNS specifies the naming structure of the hosts, as well as name-to-address resolution. Hosts on the Internet are logically addressed with multipart names known as IP addresses. The parts of an IP address progress from the most specific to the most general, with periods separating the fields. For instance, *eric.cs.yale.edu* refers to host *eric* in the Department of Computer Science at Yale University within the top-level domain *edu*. (Other top-level domains include *com* for commercial sites and *org* for organizations, as well as a domain for each country connected to the network for systems specified by country rather than organization type.) Generally, the system resolves addresses by examining the host-name components in reverse order. Each component has a **name server**—simply a process on a system—that accepts a name and returns the address of the name server responsible for that name. As the final step, the name server for the host in question is contacted, and a host-id is returned. For example, a request made by a process on system A to communicate with *eric.cs.yale.edu* would result in the following steps:

1. The system library or the kernel on system A issues a request to the name server for the `edu` domain, asking for the address of the name server for `yale.edu`. The name server for the *edu* domain must be at a known address, so that it can be queried.
2. The `edu` name server returns the address of the host on which the `yale.edu` name server resides.
3. System A then queries the name server at this address and asks about `cs.yale.edu`.
4. An address is returned. Now, finally, a request to that address for `eric.cs.yale.edu` returns an Internet address host-id for that host (for example, `128.148.31.100`).

This protocol may seem inefficient, but individual hosts cache the IP addresses they have already resolved to speed the process. (Of course, the contents of these caches must be refreshed over time in case the name server is moved

```
/**
 * Usage: java DNSLookUp <IP name>
 * i.e. java DNSLookUp www.wiley.com
 */
public class DNSLookUp {
  public static void main(String[] args) {
    InetAddress hostAddress;

    try {
      hostAddress = InetAddress.getByName(args[0]);
      System.out.println(hostAddress.getHostAddress());
    }
    catch (UnknownHostException uhe) {
      System.err.println("Unknown host: " + args[0]);
    }
  }
}
```

**Figure 19.4** Java program illustrating a DNS lookup.

or its address changes.) In fact, the protocol is so important that it has been optimized many times and has had many safeguards added. Consider what would happen if the primary `edu` name server crashed. It is possible that no `edu` hosts would be able to have their addresses resolved, making them all unreachable! The solution is to use secondary, backup name servers that duplicate the contents of the primary servers.

Before the domain-name service was introduced, all hosts on the Internet needed to have copies of a file (mentioned above) that contained the names and addresses of each host on the network. All changes to this file had to be registered at one site (host SRI-NIC), and periodically all hosts had to copy the updated file from SRI-NIC to be able to contact new systems or find hosts whose addresses had changed. Under the domain-name service, each name-server site is responsible for updating the host information for that domain. For instance, any host changes at Yale University are the responsibility of the name server for *yale.edu* and need not be reported anywhere else. DNS lookups will automatically retrieve the updated information because they will contact *yale.edu* directly. Domains may contain autonomous subdomains to further distribute the responsibility for host-name and host-id changes.

Java provides the necessary API to design a program that maps IP names to IP addresses. The program shown in Figure 19.4 is passed an IP name (such as *eric.cs.yale.edu*) on the command line and either outputs the IP address of the host or returns a message indicating that the host name could not be resolved. An `InetAddress` is a Java class representing an IP name or address. The static method `getByName()` belonging to the `InetAddress` class is passed a string representation of an IP name, and it returns the corresponding `InetAddress`. The program then invokes the `getHostAddress()` method, which internally uses DNS to look up the IP address of the designated host.

Generally, the operating system is responsible for accepting from its processes a message destined for <host name, identifier> and for transferring that message to the appropriate host. The kernel on the destination host is then responsible for transferring the message to the process named by the identifier. This process is described in Section 19.3.4.

### 19.3.2 Communication Protocols

When we are designing a communication network, we must deal with the inherent complexity of coordinating asynchronous operations communicating in a potentially slow and error-prone environment. In addition, the systems on the network must agree on a protocol or a set of protocols for determining host names, locating hosts on the network, establishing connections, and so on. We can simplify the design problem (and related implementation) by partitioning the problem into multiple layers. Each layer on one system communicates with the equivalent layer on other systems. Typically, each layer has its own protocols, and communication takes place between peer layers using a specific protocol. The protocols may be implemented in hardware or software. For instance, Figure 19.5 shows the logical communications between two computers, with the three lowest-level layers implemented in hardware.

The International Standards Organization created the Open Systems Interconnection (OSI) model for describing the various layers of networking. While these layers are not implemented in practice, they are useful for understanding how networking logically works, and we describe them below:

- **Layer 1: Physical layer**. The physical layer is responsible for handling both the mechanical and the electrical details of the physical transmission of a bit stream. At the physical layer, the communicating systems must agree on the electrical representation of a binary 0 and 1, so that when data are sent as a stream of electrical signals, the receiver is able to interpret the data

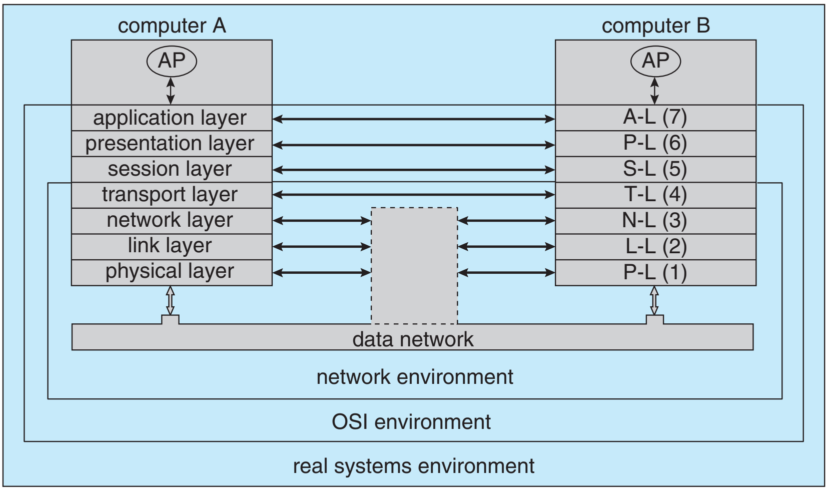

computer A
computer B
AP
AP
application layer
presentation layer
session layer
transport layer
network layer
link layer
physical layer
A-L (7)
P-L (6)
S-L (5)
T-L (4)
N-L (3)
L-L (2)
P-L (1)
data network
network environment
OSI environment
real systems environment

**Figure 19.5** Two computers communicating via the OSI network model.

properly as binary data. This layer is implemented in the hardware of the networking device. It is responsible for delivering bits.

- **Layer 2: Data-link layer**. The data-link layer is responsible for handling *frames*, or fixed-length parts of packets, including any error detection and recovery that occur in the physical layer. It sends frames between physical addresses.
- **Layer 3: Network layer**. The network layer is responsible for breaking messages into packets, providing connections between logical addresses, and routing packets in the communication network, including handling the addresses of outgoing packets, decoding the addresses of incoming packets, and maintaining routing information for proper response to changing load levels. Routers work at this layer.
- **Layer 4: Transport layer**. The transport layer is responsible for transfer of messages between nodes, maintaining packet order, and controlling flow to avoid congestion.
- **Layer 5: Session layer**. The session layer is responsible for implementing sessions, or process-to-process communication protocols.
- **Layer 6: Presentation layer**. The presentation layer is responsible for resolving the differences in formats among the various sites in the network, including character conversions and half duplex–full duplex modes (character echoing).
- **Layer 7: Application layer**. The application layer is responsible for interacting directly with users. This layer deals with file transfer, remote-login protocols, and electronic mail, as well as with schemas for distributed databases.

Figure 19.6 summarizes the **OSI protocol stack**—a set of cooperating protocols—showing the physical flow of data. As mentioned, logically each layer of a protocol stack communicates with the equivalent layer on other systems. But physically, a message starts at or above the application layer and is passed through each lower level in turn. Each layer may modify the message and include message-header data for the equivalent layer on the receiving side. Ultimately, the message reaches the data-network layer and is transferred as one or more packets (Figure 19.7). The data-link layer of the target system receives these data, and the message is moved up through the protocol stack. It is analyzed, modified, and stripped of headers as it progresses. It finally reaches the application layer for use by the receiving process.

The OSI model formalizes some of the earlier work done in network protocols but was developed in the late 1970s and is currently not in widespread use. Perhaps the most widely adopted protocol stack is the TCP/IP model (sometimes called the *Internet model*), which has been adopted by virtually all Internet sites. The TCP/IP protocol stack has fewer layers than the OSI model. Theoretically, because it combines several functions in each layer, it is more difficult to implement but more efficient than OSI networking. The relationship between the OSI and TCP/IP models is shown in Figure 19.8.

The TCP/IP application layer identifies several protocols in widespread use in the Internet, including HTTP, FTP, SSH, DNS, and SMTP. The transport layer

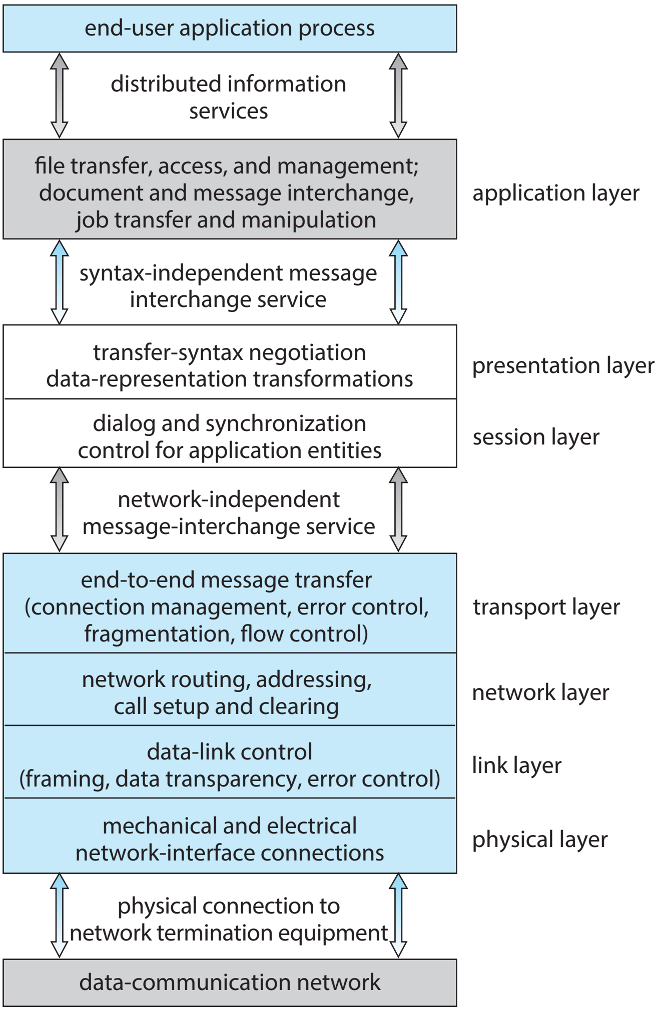

end-user application process
distributed information services
file transfer, access, and management; document and message interchange, job transfer and manipulation
application layer
syntax-independent message interchange service
transfer-syntax negotiation data-representation transformations
presentation layer
dialog and synchronization control for application entities
session layer
network-independent message-interchange service
end-to-end message transfer (connection management, error control, fragmentation, flow control)
transport layer
network routing, addressing, call setup and clearing
network layer
data-link control (framing, data transparency, error control)
link layer
mechanical and electrical network-interface connections
physical layer
physical connection to network termination equipment
data-communication network

**Figure 19.6** The OSI protocol stack.

identifies the unreliable, connectionless **user datagram protocol** (**UDP**) and the reliable, connection-oriented **transmission control protocol** (**TCP**). The **Internet protocol** (**IP**) is responsible for routing IP **datagrams**, or packets, through the Internet. The TCP/IP model does not formally identify a link or physical layer, allowing TCP/IP traffic to run across any physical network. In Section 19.3.3, we consider the TCP/IP model running over an Ethernet network.

Security should be a concern in the design and implementation of any modern communication protocol. Both strong authentication and encryption are needed for secure communication. Strong authentication ensures that the sender and receiver of a communication are who or what they are supposed to be. Encryption protects the contents of the communication from eavesdropping. Weak authentication and clear-text communication are still very common, however, for a variety of reasons. When most of the common protocols were designed, security was frequently less important than performance, sim-

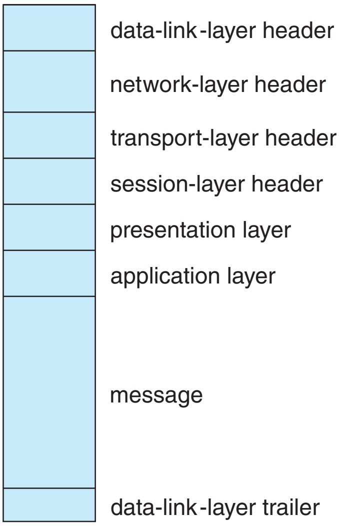

data-link-layer header
network-layer header
transport-layer header
session-layer header
presentation layer
application layer
message
data-link-layer trailer

**Figure 19.7** An OSI network message.

plicity, and efficiency. This legacy is still showing itself today, as adding security to existing infrastructure is proving to be difficult and complex.

Strong authentication requires a multistep handshake protocol or authentication devices, adding complexity to a protocol. As to the encryption requirement, modern CPUs can efficiently perform encryption, frequently including cryptographic acceleration instructions so system performance is not compromised. Long-distance communication can be made secure by authenticating

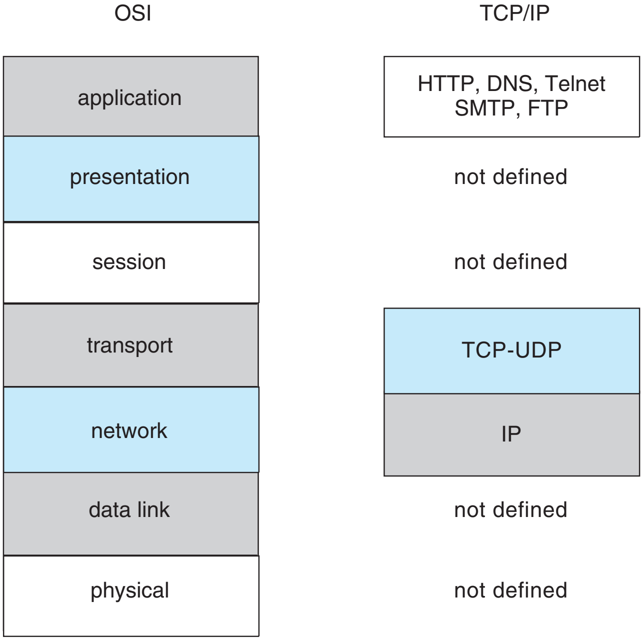

OSI
TCP/IP
application
presentation
session
transport
network
data link
physical
HTTP, DNS, Telnet SMTP, FTP
not defined
not defined
TCP-UDP
IP
not defined
not defined

**Figure 19.8** The OSI and TCP/IP protocol stacks.

the endpoints and encrypting the stream of packets in a virtual private network, as discussed in Section 16.4.2. LAN communication remains unencrypted at most sites, but protocols such as NFS Version 4, which includes strong native authentication and encryption, should help improve even LAN security.

### 19.3.3 TCP/IP Example

Next, we address name resolution and examine its operation with respect to the TCP/IP protocol stack on the Internet. Then we consider the processing needed to transfer a packet between hosts on different Ethernet networks. We base our description on the IPV4 protocols, which are the type most commonly used today.

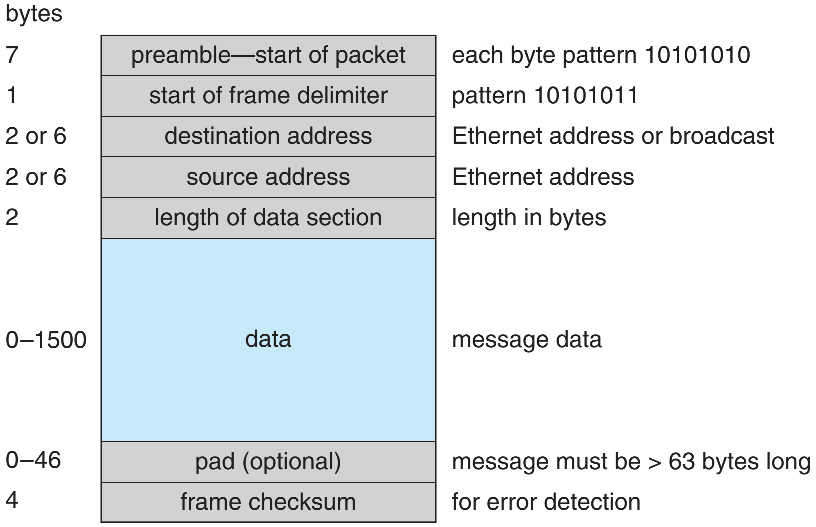
In a TCP/IP network, every host has a name and an associated IP address (or host-id). Both of these strings must be unique; and so that the name space can be managed, they are segmented. As described earlier, the name is hierarchical, describing the host name and then the organization with which the host is associated. The host-id is split into a network number and a host number. The proportion of the split varies, depending on the size of the network. Once the Internet administrators assign a network number, the site with that number is free to assign host-ids.

The sending system checks its routing tables to locate a router to send the frame on its way. This routing table is either configured manually by the system administrator or is populated by one of several routing protocols, such as the **Border Gateway Protocol** (**BGP**). The routers use the network part of the host-id to transfer the packet from its source network to the destination network. The destination system then receives the packet. The packet may be a complete message, or it may just be a component of a message, with more packets needed before the message can be reassembled and passed to the TCP/UDP (transport) layer for transmission to the destination process.

Within a network, how does a packet move from sender (host or router) to receiver? Every Ethernet device has a unique byte number, called the **medium access control** (**MAC**) **address**, assigned to it for addressing. Two devices on a LAN communicate with each other only with this number. If a system needs to send data to another system, the networking software generates an **address resolution protocol** (**ARP**) packet containing the IP address of the destination system. This packet is **broadcast** to all other systems on that Ethernet network.

A broadcast uses a special network address (usually, the maximum address) to signal that all hosts should receive and process the packet. The broadcast is not re-sent by routers in between different networks, so only systems on the local network receive it. Only the system whose IP address matches the IP address of the ARP request responds and sends back its MAC address to the system that initiated the query. For efficiency, the host caches the IP–MAC address pair in an internal table. The cache entries are aged, so that an entry is eventually removed from the cache if an access to that system is not required within a given time. In this way, hosts that are removed from a network are eventually forgotten. For added performance, ARP entries for heavily used hosts may be pinned in the ARP cache.

Once an Ethernet device has announced its host-id and address, communication can begin. A process may specify the name of a host with which to communicate. Networking software takes that name and determines the IP address of the target, using a DNS lookup or an entry in a local `hosts` file



| bytes | | |
|---|---|---|
| 7 | preamble—start of packet | each byte pattern 10101010 |
| 1 | start of frame delimiter | pattern 10101011 |
| 2 or 6 | destination address | Ethernet address or broadcast |
| 2 or 6 | source address | Ethernet address |
| 2 | length of data section | length in bytes |
| 0–1500 | data | message data |
| 0–46 | pad (optional) | message must be > 63 bytes long |
| 4 | frame checksum | for error detection |

**Figure 19.9** An Ethernet packet.

where translations can be manually stored. The message is passed from the application layer, through the software layers, and to the hardware layer. At the hardware layer, the packet has the Ethernet address at its start; a trailer indicates the end of the packet and contains a **checksum** for detection of packet damage (Figure 19.9). The packet is placed on the network by the Ethernet device. The data section of the packet may contain some or all of the data of the original message, but it may also contain some of the upper-level headers that compose the message. In other words, all parts of the original message must be sent from source to destination, and all headers above the 802.3 layer (data-link layer) are included as data in the Ethernet packets.

If the destination is on the same local network as the source, the system can look in its ARP cache, find the Ethernet address of the host, and place the packet on the wire. The destination Ethernet device then sees its address in the packet and reads in the packet, passing it up the protocol stack.

If the destination system is on a network different from that of the source, the source system finds an appropriate router on its network and sends the packet there. Routers then pass the packet along the WAN until it reaches its destination network. The router that connects the destination network checks its ARP cache, finds the Ethernet number of the destination, and sends the packet to that host. Through all of these transfers, the data-link-layer header may change as the Ethernet address of the next router in the chain is used, but the other headers of the packet remain the same until the packet is received and processed by the protocol stack and finally passed to the receiving process by the kernel.

### 19.3.4 Transport Protocols UDP and TCP

Once a host with a specific IP address receives a packet, it must somehow pass it to the correct waiting process. The transport protocols TCP and UDP identify the receiving (and sending) processes through the use of a **port number**. Thus, a host with a single IP address can have multiple server processes running and waiting for packets as long as each server process specifies a different port number. By default, many common services use *well-known* port numbers. Some examples include FTP (21), SSH (22), SMTP (25), and HTTP (80). For example, if you wish to connect to an "http" website through your web browser, your browser will automatically attempt to connect to port 80 on the server by using the number 80 as the port number in the TCP transport header. For an extensive list of well-known ports, log into your favorite Linux or UNIX machine and take a look at the file `/etc/services`.

The transport layer can accomplish more than just connecting a network packet to a running process. It can also, if desired, add reliability to a network packet stream. To explain how, we next outline some general behavior of the transport protocols UDP and TCP.

### 19.3.4.1 User Datagram Protocol

The transport protocol UDP is *unreliable* in that it is a bare-bones extension to IP with the addition of a port number. In fact, the UDP header is very simple and contains only four fields: source port number, destination port number, length, and checksum. Packets may be sent quickly to a destination using UDP. However, since there are no guarantees of delivery in the lower layers of the network stack, packets may become lost. Packets can also arrive at the receiver out of order. It is up to the application to figure out these error cases and to adjust (or not adjust).

Figure 19.10 illustrates a common scenario involving loss of a packet between a client and a server using the UDP protocol. Note that this protocol is known as a *connectionless* protocol because there is no connection setup at the beginning of the transmission to set up state—the client just starts sending data. Similarly, there is no connection teardown.

The client begins by sending some sort of request for information to the server. The server then responds by sending four datagrams, or packets, to the client. Unfortunately, one of the packets is dropped by an overwhelmed router. The client must either make do with only three packets or use logic programmed into the application to request the missing packet. Thus, we

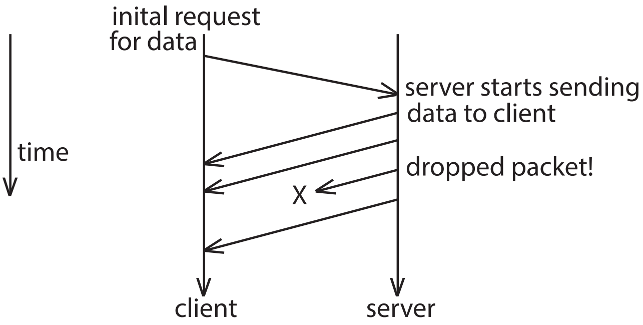


**Figure 19.10** Example of a UDP data transfer with dropped packet.

need to use a different transport protocol if we want any additional reliability guarantees to be handled by the network.

#### 19.3.4.2 Transmission Control Protocol

TCP is a transport protocol that is both *reliable* and *connection-oriented*. In addition to specifying port numbers to identify sending and receiving processes on different hosts, TCP provides an abstraction that allows a sending process on one host to send an in-order, uninterrupted *byte stream* across the network to a receiving process on another host. It accomplishes these things through the following mechanisms:

- Whenever a host sends a packet, the receiver must send an **acknowledgment packet**, or ACK, to notify the sender that the packet was received. If the ACK is not received before a timer expires, the sender will send that packet again.
- TCP introduces **sequence numbers** into the TCP header of every packet. These numbers allow the receiver to (1) put packets in order before sending data up to the requesting process and (2) be aware of packets missing from the byte stream.
- TCP connections are initiated with a series of control packets between the sender and the receiver (often called a *three-way handshake*) and closed gracefully with control packets responsible for tearing down the connection. These control packets allow both the sender and the receiver to set up and remove state.

Figure 19.11 demonstrates a possible exchange using TCP (with connection setup and tear-down omitted). After the connection has been established, the client sends a request packet to the server with the sequence number 904. Unlike the server in the UDP example, the server must then send an ACK packet back to the client. Next, the server starts sending its own stream of data packets starting with a different sequence number. The client sends an ACK packet for each data packet it receives. Unfortunately, the data packet with the sequence number 127 is lost, and no ACK packet is sent by the client. The sender times out waiting for the ACK packet, so it must resend data packet 127. Later in the connection, the server sends the data packet with the sequence number 128, but the ACK is lost. Since the server does not receive the ACK it must resend data packet 128. The client then receives a duplicate packet. Because the client knows that it previously received a packet with that sequence number, it throws the duplicate away. However, it must send another ACK back to the server to allow the server to continue.

In the actual TCP specification, an ACK isn't required for each and every packet. Instead, the receiver can send a ***cumulative ACK*** to ACK a series of packets. The server can also send numerous data packets sequentially before waiting for ACKs, to take advantage of network throughput.

TCP also helps regulate the flow of packets through mechanisms called *flow control* and *congestion control*. **Flow control** involves preventing the sender from overrunning the capacity of the receiver. For example, the receiver may

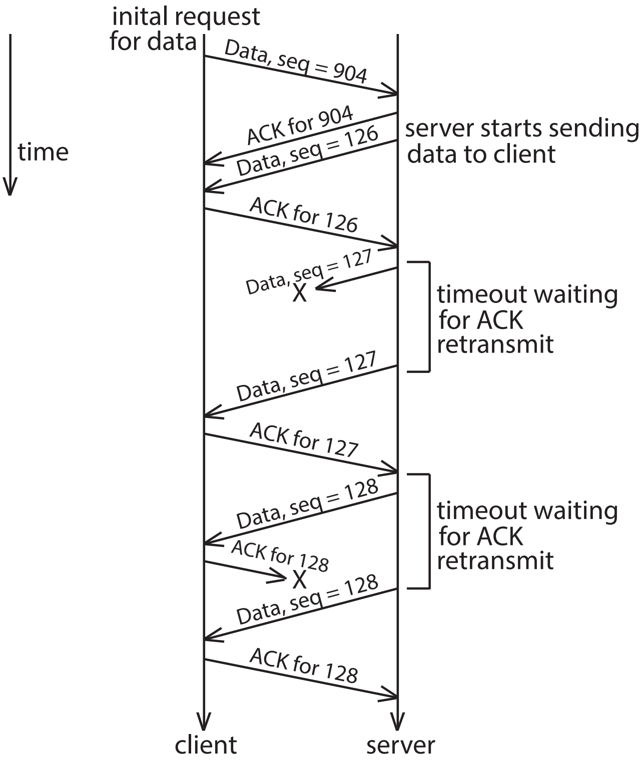

inital request
for data
time
Data, seq = 904
ACK for 904
Data, seq = 126
server starts sending
data to client
ACK for 126
Data, seq = 127
X
timeout waiting
for ACK
retransmit
Data, seq = 127
ACK for 127
Data, seq = 128
ACK for 128
X
timeout waiting
for ACK
retransmit
Data, seq = 128
ACK for 128
client
server

**Figure 19.11** Example of a TCP data transfer with dropped packets.

have a slower connection or may have slower hardware components (like a slower network card or processor). Flow-control state can be returned in the ACK packets of the receiver to alert the sender to slow down or speed up. **Congestion control** attempts to approximate the state of the networks (and generally the routers) between the sender and the receiver. If a router becomes overwhelmed with packets, it will tend to drop them. Dropping packets results in ACK timeouts, which results in more packets saturating the network. To prevent this condition, the sender monitors the connection for dropped packets by noticing how many packets are not acknowledged. If there are too many dropped packets, the sender will slow down the rate at which it sends them. This helps ensure that the TCP connection is being fair to other connections happening at the same time.

By utilizing a reliable transport protocol like TCP, a distributed system does not need extra logic to deal with lost or out-of-order packets. However, TCP is slower than UDP.

## 19.4 Network and Distributed Operating Systems

In this section, we describe the two general categories of network-oriented operating systems: network operating systems and distributed operating systems. Network operating systems are simpler to implement but generally more difficult for users to access and use than are distributed operating systems, which provide more features.

### 19.4.1 Network Operating Systems

A **network operating system** provides an environment in which users can access remote resources (implementing resource sharing) by either logging in to the appropriate remote machine or transferring data from the remote machine to their own machines. Currently, all general-purpose operating systems, and even embedded operating systems such as Android and iOS, are network operating systems.

#### 19.4.1.1 Remote Login

An important function of a network operating system is to allow users to log in remotely. The Internet provides the `ssh` facility for this purpose. To illustrate, suppose that a user at Westminster College wishes to compute on `kristen.cs.yale.edu`, a computer located at Yale University. To do so, the user must have a valid account on that machine. To log in remotely, the user issues the command

```
ssh kristen.cs.yale.edu
```

This command results in the formation of an encrypted socket connection between the local machine at Westminster College and the `kristen.cs.yale.edu` computer. After this connection has been established, the networking software creates a transparent, bidirectional link so that all characters entered by the user are sent to a process on `kristen.cs.yale.edu` and all the output from that process is sent back to the user. The process on the remote machine asks the user for a login name and a password. Once the correct information has been received, the process acts as a proxy for the user, who can compute on the remote machine just as any local user can.

#### 19.4.1.2 Remote File Transfer

Another major function of a network operating system is to provide a mechanism for **remote file transfer** from one machine to another. In such an environment, each computer maintains its own local file system. If a user at one site (say, Kurt at `albion.edu`) wants to access a file owned by Becca located on another computer (say, at `colby.edu`), then the file must be copied explicitly from the computer at Colby in Maine to the computer at Albion in Michigan. The communication is one-directional and individual, such that other users at those sites wishing to transfer a file, say Sean at `colby.edu` to Karen at `albion.edu`, must likewise issue a set of commands.

The Internet provides a mechanism for such a transfer with the file transfer protocol (FTP) and the more private secure file transfer protocol (SFTP). Suppose that user Carla at `wesleyan.edu` wants to copy a file that is owned by Owen at `kzoo.edu.` The user must first invoke the `sftp` program by executing

```
sftp owen@kzoo.edu
```

The program then asks the user for a login name and a password. Once the correct information has been received, the user can use a series of commands to upload files, download files, and navigate the remote file system structure. Some of these commands are:

- `get`—Transfer a file from the remote machine to the local machine.
- `put`—Transfer a file from the local machine to the remote machine.
- `ls` or `dir`—List files in the current directory on the remote machine.
- `cd`—Change the current directory on the remote machine.

There are also various commands to change transfer modes (for binary or ASCII files) and to determine connection status.

#### 19.4.1.3 Cloud Storage

Basic cloud-based storage applications allow users to transfer files much as with FTP. Users can upload files to a cloud server, download files to the local computer, and share files with other cloud-service users via a web link or other sharing mechanism through a graphical interface. Common examples include Dropbox and Google Drive.

An important point about SSH, FTP, and cloud-based storage applications is that they require the user to change paradigms. FTP, for example, requires the user to know a command set entirely different from the normal operating-system commands. With SSH, the user must know appropriate commands on the remote system. For instance, a user on a Windows machine who connects remotely to a UNIX machine must switch to UNIX commands for the duration of the SSH session. (In networking, a **session** is a complete round of communication, frequently beginning with a login to authenticate and ending with a logoff to terminate the communication.) With cloud-based storage applications, users may have to log into the cloud service (usually through a web browser) or native application and then use a series of graphical commands to upload, download, or share files. Obviously, users would find it more convenient not to be required to use a different set of commands. Distributed operating systems are designed to address this problem.

### 19.4.2 Distributed Operating Systems

In a distributed operating system, users access remote resources in the same way they access local resources. Data and process migration from one site to another is under the control of the distributed operating system. Depending on the goals of the system, it can implement data migration, computation migration, process migration, or any combination thereof.

#### 19.4.2.1 Data Migration

Suppose a user on site A wants to access data (such as a file) that reside at site B. The system can transfer the data by one of two basic methods. One approach to **data migration** is to transfer the entire file to site A. From that point on, all access to the file is local. When the user no longer needs access to the file, a copy of the file (if it has been modified) is sent back to site B. Even if only a modest change has been made to a large file, all the data must be transferred. This mechanism can be thought of as an automated FTP system. This approach was used in the Andrew file system, but it was found to be too inefficient.

The other approach is to transfer to site A only those portions of the file that are actually *necessary* for the immediate task. If another portion is required later, another transfer will take place. When the user no longer wants to access the file, any part of it that has been modified must be sent back to site B. (Note the similarity to demand paging.) Most modern distributed systems use this approach.

Whichever method is used, data migration includes more than the mere transfer of data from one site to another. The system must also perform various data translations if the two sites involved are not directly compatible (for instance, if they use different character-code representations or represent integers with a different number or order of bits).

#### 19.4.2.2 Computation Migration

In some circumstances, we may want to transfer the computation, rather than the data, across the system; this process is called **computation migration**. For example, consider a job that needs to access various large files that reside at different sites, to obtain a summary of those files. It would be more efficient to access the files at the sites where they reside and return the desired results to the site that initiated the computation. Generally, if the time to transfer the data is longer than the time to execute the remote command, the remote command should be used.

Such a computation can be carried out in different ways. Suppose that process P wants to access a file at site A. Access to the file is carried out at site A and could be initiated by an RPC. An RPC uses network protocols to execute a routine on a remote system (Section 3.8.2). Process P invokes a predefined procedure at site A. The procedure executes appropriately and then returns the results to P.

Alternatively, process P can send a message to site A. The operating system at site A then creates a new process Q whose function is to carry out the designated task. When process Q completes its execution, it sends the needed result back to P via the message system. In this scheme, process P may execute concurrently with process Q. In fact, it may have several processes running concurrently on several sites.

Either method could be used to access several files (or chunks of files) residing at various sites. One RPC might result in the invocation of another RPC or even in the transfer of messages to another site. Similarly, process Q could, during the course of its execution, send a message to another site, which in turn would create another process. This process might either send a message back to Q or repeat the cycle.

#### 19.4.2.3 Process Migration

A logical extension of computation migration is **process migration**. When a process is submitted for execution, it is not always executed at the site at which it is initiated. The entire process, or parts of it, may be executed at different sites. This scheme may be used for several reasons:

- **Load balancing**. The processes (or subprocesses) may be distributed across the sites to even the workload.
- **Computation speedup**. If a single process can be divided into a number of subprocesses that can run concurrently on different sites or nodes, then the total process turnaround time can be reduced.
- **Hardware preference**. The process may have characteristics that make it more suitable for execution on some specialized processor (such as matrix inversion on a GPU) than on a microprocessor.
- **Software preference**. The process may require software that is available at only a particular site, and either the software cannot be moved, or it is less expensive to move the process.
- **Data access**. Just as in computation migration, if the data being used in the computation are numerous, it may be more efficient to have a process run remotely (say, on a server that hosts a large database) than to transfer all the data and run the process locally.

We use two complementary techniques to move processes in a computer network. In the first, the system can attempt to hide the fact that the process has migrated from the client. The client then need not code her program explicitly to accomplish the migration. This method is usually employed for achieving load balancing and computation speedup among homogeneous systems, as they do not need user input to help them execute programs remotely.

The other approach is to allow (or require) the user to specify explicitly how the process should migrate. This method is usually employed when the process must be moved to satisfy a hardware or software preference.

You have probably realized that the World Wide Web has many aspects of a distributed computing environment. Certainly it provides data migration (between a web server and a web client). It also provides computation migration. For instance, a web client could trigger a database operation on a web server. Finally, with Java, Javascript, and similar languages, it provides a form of process migration: Java applets and Javascript scripts are sent from the server to the client, where they are executed. A network operating system provides most of these features, but a distributed operating system makes them seamless and easily accessible. The result is a powerful and easy-to-use facility—one of the reasons for the huge growth of the World Wide Web.

## 19.5 Design Issues in Distributed Systems

The designers of a distributed system must take a number of design challenges into account. The system should be robust so that it can withstand failures. The system should also be transparent to users in terms of both file location and user mobility. Finally, the system should be scalable to allow the addition of more computation power, more storage, or more users. We briefly introduce these issues here. In the next section, we put them in context when we describe the designs of specific distributed file systems.

### 19.5.1 Robustness

A distributed system may suffer from various types of hardware failure. The failure of a link, a host, or a site and the loss of a message are the most common types. To ensure that the system is robust, we must detect any of these failures, reconfigure the system so that computation can continue, and recover when the failure is repaired.

A system can be **fault tolerant** in that it can tolerate a certain level of failure and continue to function normally. The degree of fault tolerance depends on the design of the distributed system and the specific fault. Obviously, more fault tolerance is better.

We use the term ***fault tolerance*** in a broad sense. Communication faults, certain machine failures, storage-device crashes, and decays of storage media should all be tolerated to some extent. A **fault-tolerant system** should continue to function, perhaps in a degraded form, when faced with such failures. The degradation can affect performance, functionality, or both. It should be proportional, however, to the failures that caused it. A system that grinds to a halt when only one of its components fails is certainly not fault tolerant.

Unfortunately, fault tolerance can be difficult and expensive to implement. At the network layer, multiple redundant communication paths and network devices such as switches and routers are needed to avoid a communication failure. A storage failure can cause loss of the operating system, applications, or data. Storage units can include redundant hardware components that automatically take over from each other in case of failure. In addition, RAID systems can ensure continued access to the data even in the event of one or more storage device failures (Section 11.8).

#### 19.5.1.1 Failure Detection

In an environment with no shared memory, we generally cannot differentiate among link failure, site failure, host failure, and message loss. We can usually detect only that one of these failures has occurred. Once a failure has been detected, appropriate action must be taken. What action is appropriate depends on the particular application.

To detect link and site failure, we use a **heartbeat** procedure. Suppose that sites A and B have a direct physical link between them. At fixed intervals, the sites send each other an *I-am-up* message. If site A does not receive this message within a predetermined time period, it can assume that site B has failed, that the link between A and B has failed, or that the message from B has been lost. At this point, site A has two choices. It can wait for another time period to receive an *I-am-up* message from B, or it can send an *Are-you-up?* message to B.

If time goes by and site A still has not received an *I-am-up* message, or if site A has sent an *Are-you-up?* message and has not received a reply, the procedure can be repeated. Again, the only conclusion that site A can draw safely is that some type of failure has occurred.

Site A can try to differentiate between link failure and site failure by sending an *Are-you-up?* message to B by another route (if one exists). If and when B receives this message, it immediately replies positively. This positive reply tells A that B is up and that the failure is in the direct link between them. Since we do not know in advance how long it will take the message to travel from A to B and back, we must use a time-out scheme. At the time A sends the *Are-you-up?*

message, it specifies a time interval during which it is willing to wait for the reply from B. If A receives the reply message within that time interval, then it can safely conclude that B is up. If not, however (that is, if a time-out occurs), then A may conclude only that one or more of the following situations has occurred:

- Site B is down.
- The direct link (if one exists) from A to B is down.
- The alternative path from A to B is down.
- The message has been lost. (Although the use of a reliable transport protocol such as TCP should eliminate this concern.)

Site A cannot, however, determine which of these events has occurred.

#### 19.5.1.2 Reconfiguratio

Suppose that site A has discovered, through the mechanism just described, that a failure has occurred. It must then initiate a procedure that will allow the system to reconfigure and to continue its normal mode of operation.

- If a direct link from A to B has failed, this information must be broadcast to every site in the system, so that the various routing tables can be updated accordingly.
- If the system believes that a site has failed (because that site can no longer be reached), then all sites in the system must be notified, so that they will no longer attempt to use the services of the failed site. The failure of a site that serves as a central coordinator for some activity (such as deadlock detection) requires the election of a new coordinator. Note that, if the site has not failed (that is, if it is up but cannot be reached), then we may have the undesirable situation in which two sites serve as the coordinator. When the network is partitioned, the two coordinators (each for its own partition) may initiate conflicting actions. For example, if the coordinators are responsible for implementing mutual exclusion, we may have a situation in which two processes are executing simultaneously in their critical sections.

#### 19.5.1.3 Recovery from Failure

When a failed link or site is repaired, it must be integrated into the system gracefully and smoothly.

- Suppose that a link between A and B has failed. When it is repaired, both A and B must be notified. We can accomplish this notification by continuously repeating the heartbeat procedure described in Section 19.5.1.1.
- Suppose that site B has failed. When it recovers, it must notify all other sites that it is up again. Site B then may have to receive information from the other sites to update its local tables. For example, it may need routing-table information, a list of sites that are down, undelivered messages, a

transaction log of unexecuted transactions, and mail. If the site has not failed but simply cannot be reached, then it still needs this information.

### 19.5.2 Transparency

Making the multiple processors and storage devices in a distributed system **transparent** to the users has been a key challenge to many designers. Ideally, a distributed system should look to its users like a conventional, centralized system. The user interface of a transparent distributed system should not distinguish between local and remote resources. That is, users should be able to access remote resources as though these resources were local, and the distributed system should be responsible for locating the resources and for arranging for the appropriate interaction.

Another aspect of transparency is user mobility. It would be convenient to allow users to log into any machine in the system rather than forcing them to use a specific machine. A transparent distributed system facilitates user mobility by bringing over a user's environment (for example, home directory) to wherever he logs in. Protocols like LDAP provide an authentication system for local, remote, and mobile users. Once the authentication is complete, facilities like desktop virtualization allow users to see their desktop sessions at remote facilities.

### 19.5.3 Scalability

Still another issue is **scalability**—the capability of a system to adapt to increased service load. Systems have bounded resources and can become completely saturated under increased load. For example, with respect to a file system, saturation occurs either when a server's CPU runs at a high utilization rate or when disks' I/O requests overwhelm the I/O subsystem. Scalability is a relative property, but it can be measured accurately. A scalable system reacts more gracefully to increased load than does a nonscalable one. First, its performance degrades more moderately; and second, its resources reach a saturated state later. Even perfect design however cannot accommodate an ever-growing load. Adding new resources might solve the problem, but it might generate additional indirect load on other resources (for example, adding machines to a distributed system can clog the network and increase service loads). Even worse, expanding the system can call for expensive design modifications. A scalable system should have the potential to grow without these problems. In a distributed system, the ability to scale up gracefully is of special importance, since expanding a network by adding new machines or interconnecting two networks is commonplace. In short, a scalable design should withstand high service load, accommodate growth of the user community, and allow simple integration of added resources.

Scalability is related to fault tolerance, discussed earlier. A heavily loaded component can become paralyzed and behave like a faulty component. In addition, shifting the load from a faulty component to that component's backup can saturate the latter. Generally, having spare resources is essential for ensuring reliability as well as for handling peak loads gracefully. Thus, the multiple resources in a distributed system represent an inherent advantage, giving the system a greater potential for fault tolerance and scalability. However, inappropriate design can obscure this potential. Fault-tolerance and scalability considerations call for a design demonstrating distribution of control and data.

Scalability can also be related to efficient storage schemes. For example, many cloud storage providers use **compression** or **deduplication** to cut down on the amount of storage used. *Compression* reduces the size of a file. For example, a zip archive file can be generated out of a file (or files) by executing a `zip` command, which runs a lossless compression algorithm over the data specified. (***Lossless compression*** allows original data to be perfectly reconstructed from compressed data.) The result is a file archive that is smaller than the uncompressed file. To restore the file to its original state, a user runs some sort of `unzip` command over the zip archive file. *Deduplication* seeks to lower data storage requirements by removing redundant data. With this technology, only one instance of data is stored across an entire system (even across data owned by multiple users). Both compression and deduplication can be performed at the file level or the block level, and they can be used together. These techniques can be automatically built into a distributed system to compress information without users explicitly issuing commands, thereby saving storage space and possibly cutting down on network communication costs without adding user complexity.

## 19.6 Distributed File Systems

Although the World Wide Web is the predominant distributed system in use today, it is not the only one. Another important and popular use of distributed computing is the **distributed file system**, or **DFS**.

To explain the structure of a DFS, we need to define the terms *service*, *server*, and *client* in the DFS context. A **service** is a software entity running on one or more machines and providing a particular type of function to clients. A **server** is the service software running on a single machine. A **client** is a process that can invoke a service using a set of operations that form its **client interface**. Sometimes a lower-level interface is defined for the actual cross-machine interaction; it is the **intermachine interface**.

Using this terminology, we say that a file system provides file services to clients. A client interface for a file service is formed by a set of primitive file operations, such as create a file, delete a file, read from a file, and write to a file. The primary hardware component that a file server controls is a set of local secondary-storage devices (usually, hard disks or solid-state drives) on which files are stored and from which they are retrieved according to the clients' requests.

A DFS is a file system whose clients, servers, and storage devices are dispersed among the machines of a distributed system. Accordingly, service activity has to be carried out across the network. Instead of a single centralized data repository, the system frequently has multiple and independent storage devices. As you will see, the concrete configuration and implementation of a DFS may vary from system to system. In some configurations, servers run on dedicated machines. In others, a machine can be both a server and a client.

The distinctive features of a DFS are the multiplicity and autonomy of clients and servers in the system. Ideally, though, a DFS should appear to its clients to be a conventional, centralized file system. That is, the client interface of a DFS should not distinguish between local and remote files. It is up to the DFS to locate the files and to arrange for the transport of the data. A *transparent* DFS—like the transparent distributed systems mentioned earlier—facilitates user mobility by bringing a user's environment (for example, the user's home directory) to wherever the user logs in.

The most important performance measure of a DFS is the amount of time needed to satisfy service requests. In conventional systems, this time consists of storage-access time and a small amount of CPU-processing time. In a DFS, however, a remote access has the additional overhead associated with the distributed structure. This overhead includes the time to deliver the request to a server, as well as the time to get the response across the network back to the client. For each direction, in addition to the transfer of the information, there is the CPU overhead of running the communication protocol software. The performance of a DFS can be viewed as another dimension of the DFS's transparency. That is, the performance of an ideal DFS would be comparable to that of a conventional file system.

The basic architecture of a DFS depends on its ultimate goals. Two widely used architectural models we discuss here are the **client–server model** and the **cluster-based model**. The main goal of a client–server architecture is to allow transparent file sharing among one or more clients as if the files were stored locally on the individual client machines. The distributed file systems NFS and OpenAFS are prime examples. NFS is the most common UNIX-based DFS. It has several versions, and here we refer to NFS Version 3 unless otherwise noted.

If many applications need to be run in parallel on large data sets with high availability and scalability, the cluster-based model is more appropriate than the client–server model. Two well-known examples are the Google file system and the open-source HDFS, which runs as part of the Hadoop framework.

### 19.6.1 The Client–Server DFS Model

Figure 19.12 illustrates a simple DFS **client–server model**. The server stores both files and metadata on attached storage. In some systems, more than one server can be used to store different files. Clients are connected to the server through a network and can request access to files in the DFS by contacting the server through a well-known protocol such as NFS Version 3. The server

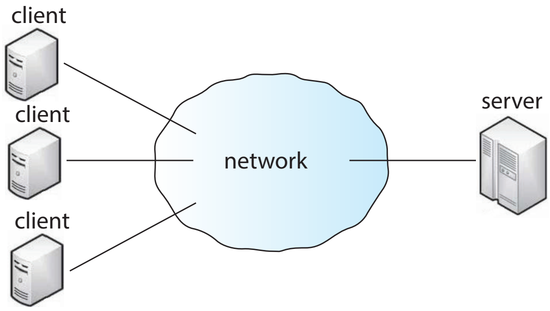


**Figure 19.12** Client–server DFS model.

is responsible for carrying out authentication, checking the requested file permissions, and, if warranted, delivering the file to the requesting client. When a client makes changes to the file, the client must somehow deliver those changes to the server (which holds the master copy of the file). The client's and the server's versions of the file should be kept consistent in a way that minimizes network traffic and the server's workload to the extent possible.

The **network file system** (**NFS**) protocol was originally developed by Sun Microsystems as an open protocol, which encouraged early adoption across different architectures and systems. From the beginning, the focus of NFS was simple and fast crash recovery in the face of server failure. To implement this goal, the NFS server was designed to be stateless; it does not keep track of which client is accessing which file or of things such as open file descriptors and file pointers. This means that, whenever a client issues a file operation (say, to read a file), that operation has to be idempotent in the face of server crashes. **Idempotent** describes an operation that can be issued more than once yet return the same result. In the case of a read operation, the client keeps track of the state (such as the file pointer) and can simply reissue the operation if the server has crashed and come back online. You can read more about the NFS implementation in Section 15.8.

The **Andrew file system** (**OpenAFS**) was created at Carnegie Mellon University with a focus on scalability. Specifically, the researchers wanted to design a protocol that would allow the server to support as many clients as possible. This meant minimizing requests and traffic to the server. When a client requests a file, the file's contents are downloaded from the server and stored on the client's local storage. Updates to the file are sent to the server when the file is closed, and new versions of the file are sent to the client when the file is opened. In comparison, NFS is quite chatty and will send block read and write requests to the server as the file is being used by a client.

Both OpenAFS and NFS are meant to be used in addition to local file systems. In other words, you would not format a hard drive partition with the NFS file system. Instead, on the server, you would format the partition with a local file system of your choosing, such as ext4, and export the shared directories via the DFS. In the client, you would simply attach the exported directories to your file-system tree. In this way, the DFS can be separated from responsibility for the local file system and can concentrate on distributed tasks.

The DFS client–server model, by design, may suffer from a single point of failure if the server crashes. Computer clustering can help resolve this problem by using redundant components and clustering methods such that failures are detected and failing over to working components continues server operations. In addition, the server presents a bottleneck for all requests for both data and metadata, which results in problems of scalability and bandwidth.

### 19.6.2 The Cluster-Based DFS Model

As the amount of data, I/O workload, and processing expands, so does the need for a DFS to be fault-tolerant and scalable. Large bottlenecks cannot be tolerated, and system component failures must be expected. Cluster-based architecture was developed in part to meet these needs.

Figure 19.13 illustrates a sample cluster-based DFS model. This is the basic model presented by the **Google file system** (**GFS**) and the **Hadoop distributed**

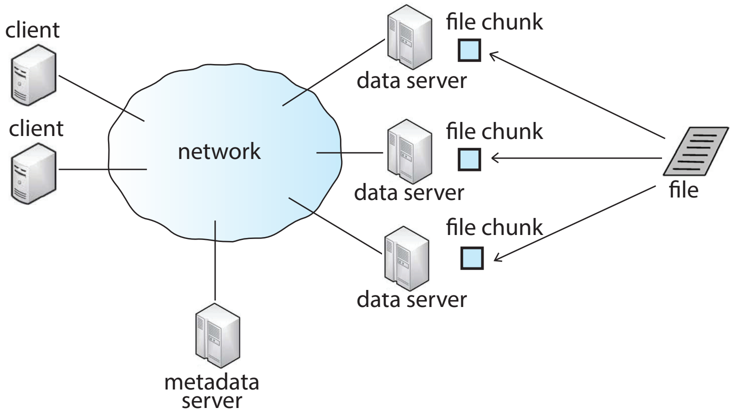


**Figure 19.13** An example of a cluster-based DFS model

**file system** (**HDFS**). One or more clients are connected via a network to a master metadata server and several data servers that house "chunks" (or portions) of files. The metadata server keeps a mapping of which data servers hold chunks of which files, as well as a traditional hierarchical mapping of directories and files. Each file chunk is stored on a data server and is replicated a certain number of times (for example, three times) to protect against component failure and for faster access to the data (servers containing the replicated chunks have fast access to those chunks).

To obtain access to a file, a client must first contact the metadata server. The metadata server then returns to the client the identities of the data servers that hold the requested file chunks. The client can then contact the closest data server (or servers) to receive the file information. Different chunks of the file can be read or written to in parallel if they are stored on different data servers, and the metadata server may need to be contacted only once in the entire process. This makes the metadata server less likely to be a performance bottleneck. The metadata server is also responsible for redistributing and balancing the file chunks among the data servers.

GFS was released in 2003 to support large distributed data-intensive applications. The design of GFS was influenced by four main observations:

- Hardware component failures are the norm rather than the exception and should be routinely expected.
- Files stored on such a system are very large.
- Most files are changed by appending new data to the end of the file rather than overwriting existing data.
- Redesigning the applications and file system API increases the system's flexibility.

Consistent with the fourth observation, GFS exports its own API and requires applications to be programmed with this API.

Shortly after developing GFS, Google developed a modularized software layer called **MapReduce** to sit on top of GFS. MapReduce allows developers to carry out large-scale parallel computations more easily and utilizes the benefits of the lower-layer file system. Later, HDFS and the Hadoop framework (which includes stackable modules like MapReduce on top of HDFS) were created based on Google's work. Like GFS and MapReduce, Hadoop supports the processing of large data sets in distributed computing environments. As suggested earlier, the drive for such a framework occurred because traditional systems could not scale to the capacity and performance needed by "big data" projects (at least not at reasonable prices). Examples of big data projects include crawling and analyzing social media, customer data, and large amounts of scientific data points for trends.

## 19.7 DFS Naming and Transparency

**Naming** is a mapping between logical and physical objects. For instance, users deal with logical data objects represented by file names, whereas the system manipulates physical blocks of data stored on disk tracks. Usually, a user refers to a file by a textual name. The latter is mapped to a lower-level numerical identifier that in turn is mapped to disk blocks. This multilevel mapping provides users with an abstraction of a file that hides the details of how and where on the disk the file is stored.

In a transparent DFS, a new dimension is added to the abstraction: that of hiding where in the network the file is located. In a conventional file system, the range of the naming mapping is an address within a disk. In a DFS, this range is expanded to include the specific machine on whose disk the file is stored. Going one step further with the concept of treating files as abstractions leads to the possibility of **file replication**. Given a file name, the mapping returns a set of the locations of this file's replicas. In this abstraction, both the existence of multiple copies and their locations are hidden.

### 19.7.1 Naming Structures

We need to differentiate two related notions regarding name mappings in a DFS:

1. **Location transparency**. The name of a file does not reveal any hint of the file's physical storage location.
2. **Location independence**. The name of a file need not be changed when the file's physical storage location changes.

Both definitions relate to the level of naming discussed previously, since files have different names at different levels (that is, user-level textual names and system-level numerical identifiers). A location-independent naming scheme is a dynamic mapping, since it can map the same file name to different locations at two different times. Therefore, location independence is a stronger property than location transparency.

In practice, most of the current DFSs provide a static, location-transparent mapping for user-level names. Some support **file migration**—that is, changing the location of a file automatically, providing location independence. OpenAFS supports location independence and file mobility, for example. HDFS includes file migration but does so without following POSIX standards, providing more flexibility in implementation and interface. HDFS keeps track of the location of data but hides this information from clients. This dynamic location transparency allows the underlying mechanism to self-tune. In another example, Amazon's S3 cloud storage facility provides blocks of storage on demand via APIs, placing the storage where it sees fit and moving the data as necessary to meet performance, reliability, and capacity requirements.

A few aspects can further differentiate location independence and static location transparency:

- Divorce of data from location, as exhibited by location independence, provides a better abstraction for files. A file name should denote the file's most significant attributes, which are its contents rather than its location. Location-independent files can be viewed as logical data containers that are not attached to a specific storage location. If only static location transparency is supported, the file name still denotes a specific, although hidden, set of physical disk blocks.
- Static location transparency provides users with a convenient way to share data. Users can share remote files by simply naming the files in a location-transparent manner, as though the files were local. Dropbox and other cloud-based storage solutions work this way. Location independence promotes sharing the storage space itself, as well as the data objects. When files can be mobilized, the overall, system-wide storage space looks like a single virtual resource. A possible benefit is the ability to balance the utilization of storage across the system.
- Location independence separates the naming hierarchy from the storage-devices hierarchy and from the intercomputer structure. By contrast, if static location transparency is used (although names are transparent), we can easily expose the correspondence between component units and machines. The machines are configured in a pattern similar to the naming structure. This configuration may restrict the architecture of the system unnecessarily and conflict with other considerations. A server in charge of a root directory is an example of a structure that is dictated by the naming hierarchy and contradicts decentralization guidelines.

Once the separation of name and location has been completed, clients can access files residing on remote server systems. In fact, these clients may be **diskless** and rely on servers to provide all files, including the operating-system kernel. Special protocols are needed for the boot sequence, however. Consider the problem of getting the kernel to a diskless workstation. The diskless workstation has no kernel, so it cannot use the DFS code to retrieve the kernel. Instead, a special boot protocol, stored in read-only memory (ROM) on the client, is invoked. It enables networking and retrieves only one special file (the kernel or boot code) from a fixed location. Once the kernel is copied over the network and loaded, its DFS makes all the other operating-system files available. The advantages of diskless clients are many, including lower cost (because the client machines require no disks) and greater convenience (when an operating-system upgrade occurs, only the server needs to be modified).

The disadvantages are the added complexity of the boot protocols and the performance loss resulting from the use of a network rather than a local disk.

### 19.7.2 Naming Schemes

There are three main approaches to naming schemes in a DFS. In the simplest approach, a file is identified by some combination of its host name and local name, which guarantees a unique system-wide name. In Ibis, for instance, a file is identified uniquely by the name *host:local-name,* where *local-name* is a UNIX-like path. The Internet URL system also uses this approach. This naming scheme is neither location transparent nor location independent. The DFS is structured as a collection of isolated component units, each of which is an entire conventional file system. Component units remain isolated, although means are provided to refer to remote files. We do not consider this scheme any further here.

The second approach was popularized by NFS. NFS provides a means to attach remote directories to local directories, thus giving the appearance of a coherent directory tree. Early NFS versions allowed only previously mounted remote directories to be accessed transparently. The advent of the **automount** feature allowed mounts to be done on demand based on a table of mount points and file-structure names. Components are integrated to support transparent sharing, but this integration is limited and is not uniform, because each machine may attach different remote directories to its tree. The resulting structure is versatile.

We can achieve total integration of the component file systems by using a third approach. Here, a single global name structure spans all the files in the system. OpenAFS provides a single global namespace for the files and directories it exports, allowing a similar user experience across different client machines. Ideally, the composed file-system structure is the same as the structure of a conventional file system. In practice, however, the many special files (for example, UNIX device files and machine-specific binary directories) make this goal difficult to attain.

To evaluate naming structures, we look at their administrative complexity. The most complex and most difficult-to-maintain structure is the NFS structure. Because any remote directory can be attached anywhere on the local directory tree, the resulting hierarchy can be highly unstructured. If a server becomes unavailable, some arbitrary set of directories on different machines becomes unavailable. In addition, a separate accreditation mechanism controls which machine is allowed to attach which directory to its tree. Thus, a user might be able to access a remote directory tree on one client but be denied access on another client.

### 19.7.3 Implementation Techniques

Implementation of transparent naming requires a provision for the mapping of a file name to the associated location. To keep this mapping manageable, we must aggregate sets of files into component units and provide the mapping on a component-unit basis rather than on a single-file basis. This aggregation serves administrative purposes as well. UNIX-like systems use the hierarchical directory tree to provide name-to-location mapping and to aggregate files recursively into directories.

To enhance the availability of the crucial mapping information, we can use replication, local caching, or both. As we noted, location independence means that the mapping changes over time. Hence, replicating the mapping makes a simple yet consistent update of this information impossible. To overcome this obstacle, we can introduce low-level, ***location-independent file identifiers***. (OpenAFS uses this approach.) Textual file names are mapped to lower-level file identifiers that indicate to which component unit the file belongs. These identifiers are still location independent. They can be replicated and cached freely without being invalidated by migration of component units. The inevitable price is the need for a second level of mapping, which maps component units to locations and needs a simple yet consistent update mechanism. Implementing UNIX-like directory trees using these low-level, location-independent identifiers makes the whole hierarchy invariant under component-unit migration. The only aspect that does change is the component-unit location mapping.

A common way to implement low-level identifiers is to use structured names. These names are bit strings that usually have two parts. The first part identifies the component unit to which the file belongs; the second part identifies the particular file within the unit. Variants with more parts are possible. The invariant of structured names, however, is that individual parts of the name are unique at all times only within the context of the rest of the parts. We can obtain uniqueness at all times by taking care not to reuse a name that is still in use, by adding sufficiently more bits (this method is used in OpenAFS), or by using a timestamp as one part of the name (as was done in Apollo Domain). Another way to view this process is that we are taking a location-transparent system, such as Ibis, and adding another level of abstraction to produce a location-independent naming scheme.

## 19.8 Remote File Access

Next, let's consider a user who requests access to a remote file. The server storing the file has been located by the naming scheme, and now the actual data transfer must take place.

One way to achieve this transfer is through a **remote-service mechanism**, whereby requests for accesses are delivered to the server, the server machine performs the accesses, and their results are forwarded back to the user. One of the most common ways of implementing remote service is the RPC paradigm, which we discussed in Chapter 3. A direct analogy exists between disk-access methods in conventional file systems and the remote-service method in a DFS: using the remote-service method is analogous to performing a disk access for each access request.

To ensure reasonable performance of a remote-service mechanism, we can use a form of caching. In conventional file systems, the rationale for caching is to reduce disk I/O (thereby increasing performance), whereas in DFSs, the goal is to reduce both network traffic and disk I/O. In the following discussion, we describe the implementation of caching in a DFS and contrast it with the basic remote-service paradigm.

### 19.8.1 Basic Caching Scheme

The concept of caching is simple. If the data needed to satisfy the access request are not already cached, then a copy of the data is brought from the server to the client system. Accesses are performed on the cached copy. The idea is to retain recently accessed disk blocks in the cache, so that repeated accesses to the same information can be handled locally, without additional network traffic. A replacement policy (for example, the least-recently-used algorithm) keeps the cache size bounded. No direct correspondence exists between accesses and traffic to the server. Files are still identified with one master copy residing at the server machine, but copies (or parts) of the file are scattered in different caches. When a cached copy is modified, the changes need to be reflected on the master copy to preserve the relevant consistency semantics. The problem of keeping the cached copies consistent with the master file is the **cache-consistency problem**, which we discuss in Section 19.8.4. DFS caching could just as easily be called **network virtual memory**. It acts similarly to demand-paged virtual memory, except that the backing store usually is a remote server rather than a local disk. NFS allows the swap space to be mounted remotely, so it actually can implement virtual memory over a network, though with a resulting performance penalty.

The granularity of the cached data in a DFS can vary from blocks of a file to an entire file. Usually, more data are cached than are needed to satisfy a single access, so that many accesses can be served by the cached data. This procedure is much like disk read-ahead (Section 14.6.2). OpenAFS caches files in large chunks (64 KB). The other systems discussed here support caching of individual blocks driven by client demand. Increasing the caching unit increases the hit ratio, but it also increases the miss penalty, because each miss requires more data to be transferred. It increases the potential for consistency problems as well. Selecting the unit of caching involves considering parameters such as the network transfer unit and the RPC protocol service unit (if an RPC protocol is used). The network transfer unit (for Ethernet, a packet) is about 1.5 KB, so larger units of cached data need to be disassembled for delivery and reassembled on reception.

Block size and total cache size are obviously of importance for block-caching schemes. In UNIX-like systems, common block sizes are 4 KB and 8 KB. For large caches (over 1 MB), large block sizes (over 8 KB) are beneficial. For smaller caches, large block sizes are less beneficial because they result in fewer blocks in the cache and a lower hit ratio.

### 19.8.2 Cache Location

Where should the cached data be stored—on disk or in main memory? Disk caches have one clear advantage over main-memory caches: they are reliable. Modifications to cached data are lost in a crash if the cache is kept in volatile memory. Moreover, if the cached data are kept on disk, they are still there during recovery, and there is no need to fetch them again. Main-memory caches have several advantages of their own, however:

- Main-memory caches permit workstations to be diskless.
- Data can be accessed more quickly from a cache in main memory than from one on a disk.
- Technology is moving toward larger and less expensive memory. The resulting performance speedup is predicted to outweigh the advantages of disk caches.

- The server caches (used to speed up disk I/O) will be in main memory regardless of where user caches are located; if we use main-memory caches on the user machine, too, we can build a single caching mechanism for use by both servers and users.

Many remote-access implementations can be thought of as hybrids of caching and remote service. In NFS, for instance, the implementation is based on remote service but is augmented with client- and server-side memory caching for performance. Thus, to evaluate the two methods, we must evaluate the degree to which either method is emphasized. The NFS protocol and most implementations do not provide disk caching (but OpenAFS does).

### 19.8.3 Cache-Update Policy

The policy used to write modified data blocks back to the server's master copy has a critical effect on the system's performance and reliability. The simplest policy is to write data through to disk as soon as they are placed in any cache. The advantage of a **write-through policy** is reliability: little information is lost when a client system crashes. However, this policy requires each write access to wait until the information is sent to the server, so it causes poor write performance. Caching with write-through is equivalent to using remote service for write accesses and exploiting caching only for read accesses.

An alternative is the **delayed-write policy**, also known as **write-back caching**, where we delay updates to the master copy. Modifications are written to the cache and then are written through to the server at a later time. This policy has two advantages over write-through. First, because writes are made to the cache, write accesses complete much more quickly. Second, data may be overwritten before they are written back, in which case only the last update needs to be written at all. Unfortunately, delayed-write schemes introduce reliability problems, since unwritten data are lost whenever a user machine crashes.

Variations of the delayed-write policy differ in when modified data blocks are flushed to the server. One alternative is to flush a block when it is about to be ejected from the client's cache. This option can result in good performance, but some blocks can reside in the client's cache a long time before they are written back to the server. A compromise between this alternative and the write-through policy is to scan the cache at regular intervals and to flush blocks that have been modified since the most recent scan, just as UNIX scans its local cache. NFS uses the policy for file data, but once a write is issued to the server during a cache flush, the write must reach the server's disk before it is considered complete. NFS treats metadata (directory data and file-attribute data) differently. Any metadata changes are issued synchronously to the server. Thus, file-structure loss and directory-structure corruption are avoided when a client or the server crashes.

Yet another variation on delayed write is to write data back to the server when the file is closed. This **write-on-close policy** is used in OpenAFS. In the case of files that are open for short periods or are modified rarely, this policy does not significantly reduce network traffic. In addition, the write-on-close policy requires the closing process to delay while the file is written through, which reduces the performance advantages of delayed writes. For files that are open for long periods and are modified frequently, however, the performance advantages of this policy over delayed write with more frequent flushing are apparent.

### 19.8.4 Consistency

A client machine is sometimes faced with the problem of deciding whether a locally cached copy of data is consistent with the master copy (and hence can be used). If the client machine determines that its cached data are out of date, it must cache an up-to-date copy of the data before allowing further accesses. There are two approaches to verifying the validity of cached data:

1. **Client-initiated approach**. The client initiates a validity check in which it contacts the server and checks whether the local data are consistent with the master copy. The frequency of the validity checking is the crux of this approach and determines the resulting consistency semantics. It can range from a check before every access to a check only on first access to a file (on file open, basically). Every access coupled with a validity check is delayed, compared with an access served immediately by the cache. Alternatively, checks can be initiated at fixed time intervals. Depending on its frequency, the validity check can load both the network and the server.
2. **Server-initiated approach**. The server records, for each client, the files (or parts of files) that it caches. When the server detects a potential inconsistency, it must react. A potential for inconsistency occurs when two different clients in conflicting modes cache a file. If UNIX semantics (Section 15.7) is implemented, we can resolve the potential inconsistency by having the server play an active role. The server must be notified whenever a file is opened, and the intended mode (read or write) must be indicated for every open. The server can then act when it detects that a file has been opened simultaneously in conflicting modes by disabling caching for that particular file. Actually, disabling caching results in switching to a remote-service mode of operation.

In a cluster-based DFS, the cache-consistency issue is made more complicated by the presence of a metadata server and several replicated file data chunks across several data servers. Using our earlier examples of HDFS and GFS, we can compare some differences. HDFS allows append-only write operations (no random writes) and a single file writer, while GFS does allow random writes with concurrent writers. This greatly complicates write consistency guarantees for GFS while simplifying them for HDFS.

## 19.9 Final Thoughts on Distributed File Systems

The line between DFS client–server and cluster-based architectures is blurring. The NFS Version 4.1 specification includes a protocol for a parallel version of NFS called pNFS, but as of this writing, adoption is slow.

GFS, HDFS, and other large-scale DFSs export a non-POSIX API, so they cannot transparently map directories to regular user machines as NFS and OpenAFS do. Rather, for systems to access these DFSs, they need client code installed. However, other software layers are rapidly being developed to allow NFS to be mounted on top of such DFSs. This is attractive, as it would take advantage of the scalability and other advantages of cluster-based DFSs while still allowing native operating-system utilities and users to access files directly on the DFS.

As of this writing, the open-source HDFS NFS Gateway supports NFS Version 3 and works as a proxy between HDFS and the NFS server software. Since HDFS currently does not support random writes, the HDFS NFS Gateway also does not support this capability. That means a file must be deleted and recreated from scratch even if only one byte is changed. Commercial organizations and researchers are addressing this problem and building stackable frameworks that allow stacking of a DFS, parallel computing modules (such as MapReduce), distributed databases, and exported file volumes through NFS.

One other type of file system, less complex than a cluster-based DFS but more complex than a client–server DFS, is a **clustered file system** (**CFS**) or **parallel file system** (**PFS**). A CFS typically runs over a LAN. These systems are important and widely used and thus deserve mention here, though we do not cover them in detail. Common CFSs include **Lustre** and **GPFS**, although there are many others. A CFS essentially treats $N$ systems storing data and $Y$ systems accessing that data as a single client–server instance. Whereas NFS, for example, has per-server naming, and two separate NFS servers generally provide two different naming schemes, a CFS knits various storage contents on various storage devices on various servers into a uniform, transparent name space. GPFS has its own file-system structure, but Lustre uses existing file systems such as ZFS for file storage and management. To learn more, see http://lustre.org.

Distributed file systems are in common use today, providing file sharing within LANs, within cluster environments, and across WANs. The complexity of implementing such a system should not be underestimated, especially considering that the DFS must be operating-system independent for widespread adoption and must provide availability and good performance in the presence of long distances, commodity hardware failures, sometimes frail networking, and ever-increasing users and workloads.

## 19.10 Summary

- A distributed system is a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory, and the processors communicate with one another through various communication lines, such as high-speed buses and the Internet. The processors in a distributed system vary in size and function.

- A distributed system provides the user with access to all system resources. Access to a shared resource can be provided by data migration, computation migration, or process migration. The access can be specified by the user or implicitly supplied by the operating system and applications.

- Protocol stacks, as specified by network layering models, add information to a message to ensure that it reaches its destination.
- A naming system (such as DNS) must be used to translate from a host name to a network address, and another protocol (such as ARP) may be needed to translate the network number to a network device address (an Ethernet address, for instance).
- If systems are located on separate networks, routers are needed to pass packets from source network to destination network.
- The transport protocols UDP and TCP direct packets to waiting processes through the use of unique system-wide port numbers. In addition, the TCP protocol allows the flow of packets to become a reliable, connection-oriented byte stream.
- There are many challenges to overcome for a distributed system to work correctly. Issues include naming of nodes and processes in the system, fault tolerance, error recovery, and scalability. Scalability issues include handling increased load, being fault tolerant, and using efficient storage schemes, including the possibility of compression and/or deduplication.
- A DFS is a file-service system whose clients, servers, and storage devices are dispersed among the sites of a distributed system. Accordingly, service activity has to be carried out across the network; instead of a single centralized data repository, there are multiple independent storage devices.
- There are two main types of DFS models: the client–server model and the cluster-based model. The client-server model allows transparent file sharing among one or more clients. The cluster-based model distributes the files among one or more data servers and is built for large-scale parallel data processing.
- Ideally, a DFS should look to its clients like a conventional, centralized file system (although it may not conform exactly to traditional file-system interfaces such as POSIX). The multiplicity and dispersion of its servers and storage devices should be transparent. A transparent DFS facilitates client mobility by bringing the client's environment to the site where the client logs in.
- There are several approaches to naming schemes in a DFS. In the simplest approach, files are named by some combination of their host name and local name, which guarantees a unique system-wide name. Another approach, popularized by NFS, provides a means to attach remote directories to local directories, thus giving the appearance of a coherent directory tree.
- Requests to access a remote file are usually handled by two complementary methods. With remote service, requests for accesses are delivered to the server. The server machine performs the accesses, and the results are forwarded back to the client. With caching, if the data needed to satisfy the access request are not already cached, then a copy of the data is brought from the server to the client. Accesses are performed on the cached copy. The problem of keeping the cached copies consistent with the master file is the cache-consistency problem.

## Practice Exercises

**19.1** Why would it be a bad idea for routers to pass broadcast packets between networks? What would be the advantages of doing so?

**19.2** Discuss the advantages and disadvantages of caching name translations for computers located in remote domains.

**19.3** What are two formidable problems that designers must solve to implement a network system that has the quality of transparency?

**19.4** To build a robust distributed system, you must know what kinds of failures can occur.

a. List three possible types of failure in a distributed system.

b. Specify which of the entries in your list also are applicable to a centralized system.

**19.5** Is it always crucial to know that the message you have sent has arrived at its destination safely? If your answer is "yes," explain why. If your answer is "no," give appropriate examples.

**19.6** A distributed system has two sites, A and B. Consider whether site A can distinguish among the following:

a. B goes down.

b. The link between A and B goes down.

c. B is extremely overloaded, and its response time is 100 times longer than normal.

What implications does your answer have for recovery in distributed systems?

## Further Reading

[Peterson and Davie (2012)] and [Kurose and Ross (2017)] provide general overviews of computer networks. The Internet and its protocols are described in [Comer (2000)]. Coverage of TCP/IP can be found in [Fall and Stevens (2011)] and [Stevens (1995)]. UNIX network programming is described thoroughly in [Steven et al. (2003)].

Ethernet and WiFi standards and speeds are evolving quickly. Current IEEE 802.3 Ethernet standards can be found at http://standards.ieee.org/about/get/802/802.3.html. Current IEEE 802.11 Wireless LAN standards can be found at http://standards.ieee.org/about/get/802/802.11.html.

Sun's network file system (NFS) is described by [Callaghan (2000)]. Information about OpenAFS is available from http://www.openafs.org.

Information on the Google file system can be found in [Ghemawat et al. (2003)]. The Google MapReduce method is described in http://research.google.com/archive/mapreduce.html. The Hadoop distributed file system is discussed in [K. Shvachko and Chansler (2010)], and the Hadoop framework is discussed in http://hadoop.apache.org/.

To learn more about Lustre, see http://lustre.org.

# Bibliography

**[Callaghan (2000)]** B. Callaghan, *NFS Illustrated*, Addison-Wesley (2000).

**[Comer (2000)]** D. Comer, *Internetworking with TCP/IP, Volume I,* Fourth Edition, Prentice Hall (2000).

**[Fall and Stevens (2011)]** K. Fall and R. Stevens, *TCP/IP Illustrated, Volume 1: The Protocols,* Second Edition, John Wiley and Sons (2011).

**[Ghemawat et al. (2003)]** S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System", *Proceedings of the ACM Symposium on Operating Systems Principles* (2003).

**[K. Shvachko and Chansler (2010)]** S. R. K. Shvachko, H. Kuang and R. Chansler, "The Hadoop Distributed File System" (2010).

**[Kurose and Ross (2017)]** J. Kurose and K. Ross, *Computer Networking—A Top–Down Approach,* Seventh Edition, Addison-Wesley (2017).

**[Peterson and Davie (2012)]** L. L. Peterson and B. S. Davie, *Computer Networks: A Systems Approacm,* Fifth Edition, Morgan Kaufmann (2012).

**[Steven et al. (2003)]** R. Steven, B. Fenner, and A. Rudoff, *Unix Network Programming, Volume 1: The Sockets Networking API,* Third Edition, John Wiley and Sons (2003).

**[Stevens (1995)]** R. Stevens, *TCP/IP Illustrated, Volume 2: The Implementation*, Addison-Wesley (1995).

## Chapter 19 Exercises

**19.7** What is the difference between computation migration and process migration? Which is easier to implement, and why?

**19.8** Even though the OSI model of networking specifies seven layers of functionality, most computer systems use fewer layers to implement a network. Why do they use fewer layers? What problems could the use of fewer layers cause?

**19.9** Explain why doubling the speed of the systems on an Ethernet segment may result in decreased network performance when the UDP transport protocol is used. What changes could help solve this problem?

**19.10** What are the advantages of using dedicated hardware devices for routers? What are the disadvantages of using these devices compared with using general-purpose computers?

**19.11** In what ways is using a name server better than using static host tables? What problems or complications are associated with name servers? What methods could you use to decrease the amount of traffic name servers generate to satisfy translation requests?

**19.12** Name servers are organized in a hierarchical manner. What is the purpose of using a hierarchical organization?

**19.13** The lower layers of the OSI network model provide datagram service, with no delivery guarantees for messages. A transport-layer protocol such as TCP is used to provide reliability. Discuss the advantages and disadvantages of supporting reliable message delivery at the lowest possible layer.

**19.14** Run the program shown in Figure 19.4 and determine the IP addresses of the following host names:

- www.wiley.com
- www.cs.yale.edu
- www.apple.com
- www.westminstercollege.edu
- www.ietf.org

**19.15** A DNS name can map to multiple servers, such as www.google.com. However, if we run the program shown in Figure 19.4, we get only one IP address. Modify the program to display all the server IP addresses instead of just one.

**19.16** The original HTTP protocol used TCP/IP as the underlying network protocol. For each page, graphic, or applet, a separate TCP session was constructed, used, and torn down. Because of the overhead of building and destroying TCP/IP connections, performance problems resulted from this implementation method. Would using UDP rather than TCP be a good alternative? What other changes could you make to improve HTTP performance?

**19.17** What are the advantages and the disadvantages of making the computer network transparent to the user?

**19.18** What are the benefits of a DFS compared with a file system in a centralized system?

**19.19** For each of the following workloads, identify whether a cluster-based or a client–server DFS model would handle the workload best. Explain your answers.

- Hosting student files in a university lab.
- Processing data sent by the Hubble telescope.
- Sharing data with multiple devices from a home server.

**19.20** Discuss whether OpenAFS and NFS provide the following: (a) location transparency and (b) location independence.

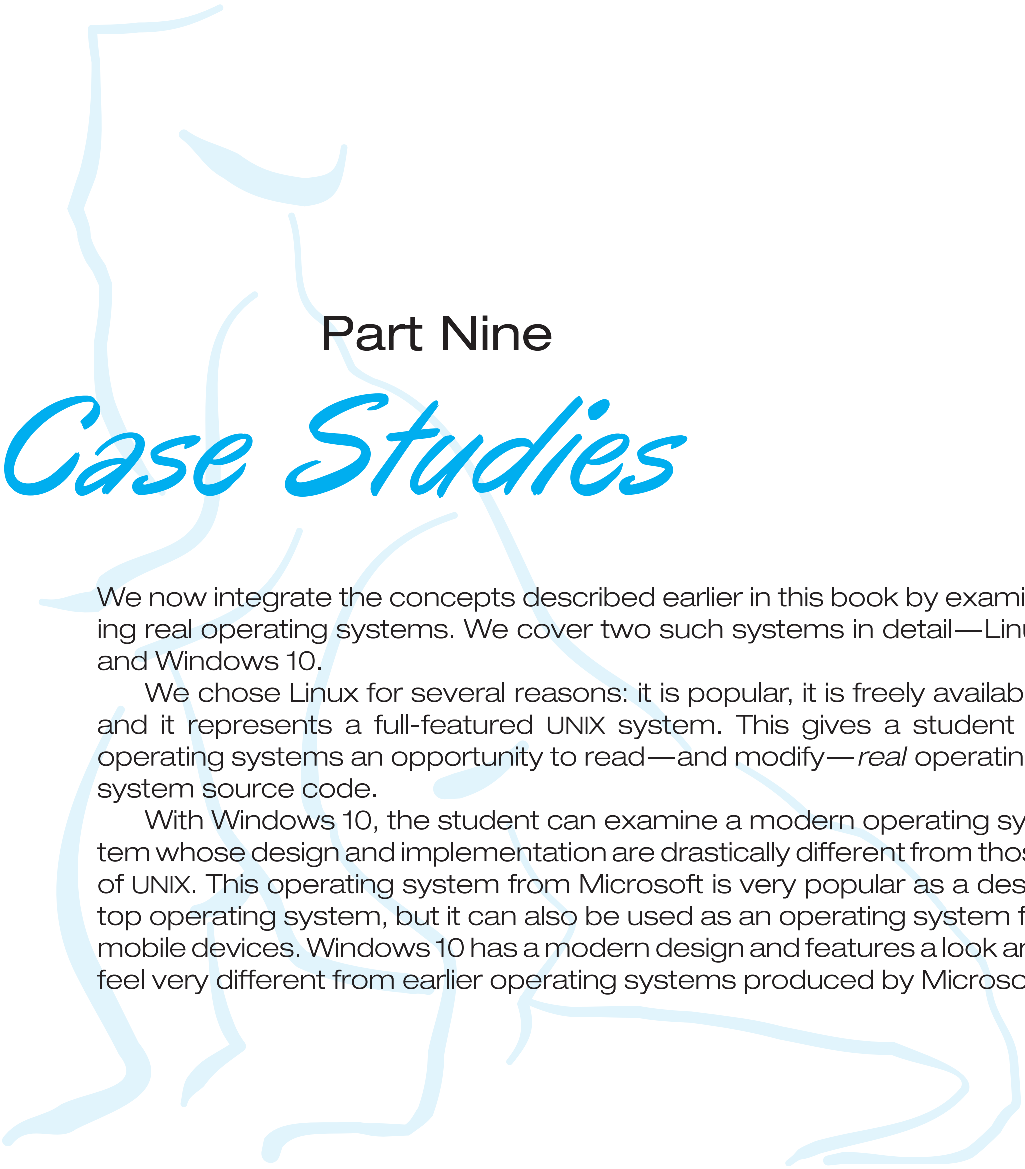
**19.21** Under what circumstances would a client prefer a location-transparent DFS? Under what circumstances would she prefer a location-independent DFS? Discuss the reasons for these preferences.

**19.22** What aspects of a distributed system would you select for a system running on a totally reliable network?

**19.23** Compare and contrast the techniques of caching disk blocks locally, on a client system, and remotely, on a server.

**19.24** Which scheme would likely result in a greater space saving on a multiuser DFS: file-level deduplication or block-level deduplication? Explain your answer.

**19.25** What types of extra metadata information would need to be stored in a DFS that uses deduplication?



# Part Nine

# Case Studies

We now integrate the concepts described earlier in this book by examining real operating systems. We cover two such systems in detail—Linux and Windows 10.

We chose Linux for several reasons: it is popular, it is freely available, and it represents a full-featured UNIX system. This gives a student of operating systems an opportunity to read—and modify—*real* operating-system source code.

With Windows 10, the student can examine a modern operating system whose design and implementation are drastically different from those of UNIX. This operating system from Microsoft is very popular as a desktop operating system, but it can also be used as an operating system for mobile devices. Windows 10 has a modern design and features a look and feel very different from earlier operating systems produced by Microsoft.

# CHAPTER 20

# The Linux System

**Updated by Robert Love**

This chapter presents an in-depth examination of the Linux operating system. By examining a complete, real system, we can see how the concepts we have discussed relate both to one another and to practice.

Linux is a variant of UNIX that has gained popularity over the last several decades, powering devices as small as mobile phones and as large as room-filling supercomputers. In this chapter, we look at the history and development of Linux and cover the user and programmer interfaces that Linux presents—interfaces that owe a great deal to the UNIX tradition. We also discuss the design and implementation of these interfaces. Linux is a rapidly evolving operating system. This chapter describes developments through the Linux 4.12 kernel, which was released in 2017.

## CHAPTER OBJECTIVES

- Explore the history of the UNIX operating system from which Linux is derived and the principles upon which Linux's design is based.
- Examine the Linux process and thread models and illustrate how Linux schedules threads and provides interprocess communication.
- Look at memory management in Linux.
- Explore how Linux implements file systems and manages I/O devices.

## 20.1 Linux History

Linux looks and feels much like any other UNIX system; indeed, UNIX compatibility has been a major design goal of the Linux project. However, Linux is much younger than most UNIX systems. Its development began in 1991, when a Finnish university student, Linus Torvalds, began creating a small but self-contained kernel for the 80386 processor, the first true 32-bit processor in Intel's range of PC-compatible CPUs.

Early in its development, the Linux source code was made available free—both at no cost and with minimal distributional restrictions—on the Internet. As a result, Linux's history has been one of collaboration by many developers from all around the world, corresponding almost exclusively over the Internet. From an initial kernel that partially implemented a small subset of the UNIX system services, the Linux system has grown to include all of the functionality expected of a modern UNIX system.

In its early days, Linux development revolved largely around the central operating-system kernel—the core, privileged executive that manages all system resources and interacts directly with the computer hardware. We need much more than this kernel, of course, to produce a full operating system. We thus need to make a distinction between the Linux kernel and a complete Linux system. The **Linux kernel** is an original piece of software developed from scratch by the Linux community. The **Linux system**, as we know it today, includes a multitude of components, some written from scratch, others borrowed from other development projects, and still others created in collaboration with other teams.

The basic Linux system is a standard environment for applications and user programming, but it does not enforce any standard means of managing the available functionality as a whole. As Linux has matured, a need has arisen for another layer of functionality on top of the Linux system. This need has been met by various Linux distributions. A **Linux distribution** includes all the standard components of the Linux system, plus a set of administrative tools to simplify the initial installation and subsequent upgrading of Linux and to manage installation and removal of other packages on the system. A modern distribution also typically includes tools for management of file systems, creation and management of user accounts, administration of networks, web browsers, word processors, and so on.

### 20.1.1 The Linux Kernel

The first Linux kernel released to the public was version 0.01, dated May 14, 1991. It had no networking, ran only on 80386-compatible Intel processors and PC hardware, and had extremely limited device-driver support. The virtual memory subsystem was also fairly basic and included no support for memory-mapped files; however, even this early incarnation supported shared pages with copy-on-write and protected address spaces. The only file system supported was the Minix file system, as the first Linux kernels were cross-developed on a Minix platform.

The next milestone, Linux 1.0, was released on March 14, 1994. This release culminated three years of rapid development of the Linux kernel. Perhaps the single biggest new feature was networking: 1.0 included support for UNIX's standard TCP/IP networking protocols, as well as a BSD-compatible socket interface for networking programming. Device-driver support was added for running IP over Ethernet or (via the PPP or SLIP protocols) over serial lines or modems.

The 1.0 kernel also included a new, much enhanced file system without the limitations of the original Minix file system, and it supported a range of SCSI controllers for high-performance disk access. The developers extended the virtual memory subsystem to support paging to swap files and memory mapping of arbitrary files (but only read-only memory mapping was implemented in 1.0).

A range of extra hardware support was included in this release. Although still restricted to the Intel PC platform, hardware support had grown to include floppy-disk and CD-ROM devices, as well as sound cards, a range of mice, and international keyboards. Floating-point emulation was provided in the kernel for 80386 users who had no 80387 math coprocessor. System V UNIX-style **interprocess communication** (**IPC**), including shared memory, semaphores, and message queues, was implemented.

At this point, development started on the 1.1 kernel stream, but numerous bug-fix patches were released subsequently for 1.0. A pattern was adopted as the standard numbering convention for Linux kernels. Kernels with an odd minor-version number, such as 1.1 or 2.5, are **development kernels**; even-numbered minor-version numbers are stable **production kernels**. Updates for the stable kernels are intended only as remedial versions, whereas the development kernels may include newer and relatively untested functionality. As we will see, this pattern remained in effect until version 3.

In March 1995, the 1.2 kernel was released. This release did not offer nearly the same improvement in functionality as the 1.0 release, but it did support a much wider variety of hardware, including the new PCI hardware bus architecture. Developers added another PC-specific feature—support for the 80386 CPU's virtual 8086 mode—to allow emulation of the DOS operating system for PC computers. They also updated the IP implementation with support for accounting and firewalling. Simple support for dynamically loadable and unloadable kernel modules was supplied as well.

The 1.2 kernel was the final PC-only Linux kernel. The source distribution for Linux 1.2 included partially implemented support for SPARC, Alpha, and MIPS CPUs, but full integration of these other architectures did not begin until after the stable 1.2 kernel was released.

The Linux 1.2 release concentrated on wider hardware support and more complete implementations of existing functionality. Much new functionality was under development at the time, but integration of the new code into the main kernel source code was deferred until after the stable 1.2 kernel was released. As a result, the 1.3 development stream saw a great deal of new functionality added to the kernel.

This work was released in June 1996 as Linux version 2.0. This release was given a major version-number increment because of two major new capabilities: support for multiple architectures, including a 64-bit native Alpha port, and symmetric multiprocessing (SMP) support. Additionally, the memory-management code was substantially improved to provide a unified cache for file-system data independent of the caching of block devices. As a result of this change, the kernel offered greatly increased file-system and virtual-memory performance. For the first time, file-system caching was extended to networked file systems, and writable memory-mapped regions were also supported. Other major improvements included the addition of internal kernel threads, a mechanism exposing dependencies between loadable modules, support for the automatic loading of modules on demand, file-system quotas, and POSIX-compatible real-time process-scheduling classes.

Improvements continued with the release of Linux 2.2 in 1999. A port to UltraSPARC systems was added. Networking was enhanced with more flexible firewalling, improved routing and traffic management, and support for TCP large window and selective acknowledgement. Acorn, Apple, and NT disks could now be read, and NFS was enhanced with a new kernel-mode NFS daemon. Signal handling, interrupts, and some I/O were locked at a finer level than before to improve symmetric multiprocessor (SMP) performance.

Advances in the 2.4 and 2.6 releases of the kernel included increased support for SMP systems, journaling file systems, and enhancements to the memory-management and block I/O systems. The thread scheduler was modified in version 2.6, providing an efficient $O(1)$ scheduling algorithm. In addition, the 2.6 kernel was preemptive, allowing a threads to be preempted even while running in kernel mode.

Linux kernel version 3.0 was released in July 2011. The major version bump from 2 to 3 occurred to commemorate the twentieth anniversary of Linux. New features include improved virtualization support, a new page write-back facility, improvements to the memory-management system, and yet another new thread scheduler—the Completely Fair Scheduler (CFS).

Linux kernel version 4.0 was released in April 2015. This time the major version bump was entirely arbitrary; Linux kernel developers simply grew tired of ever-larger minor versions. Today Linux kernel versions do not signify anything other than release ordering. The 4.0 kernel series provided support for new architectures, improved mobile functionality, and many iterative improvements. We focus on this newest kernel in the remainder of this chapter.

### 20.1.2 The Linux System

As we noted earlier, the Linux kernel forms the core of the Linux project, but other components make up a complete Linux operating system. Whereas the Linux kernel is composed entirely of code written from scratch specifically for the Linux project, much of the supporting software that makes up the Linux system is not exclusive to Linux but is common to a number of UNIX-like operating systems. In particular, Linux uses many tools developed as part of Berkeley's BSD operating system, MIT's X Window System, and the Free Software Foundation's GNU project.

This sharing of tools has worked in both directions. The main system libraries of Linux were originated by the GNU project, but the Linux community greatly improved the libraries by addressing omissions, inefficiencies, and bugs. Other components, such as the **GNU C compiler** (`gcc`), were already of sufficiently high quality to be used directly in Linux. The network administration tools under Linux were derived from code first developed for 4.3 BSD, but more recent BSD derivatives, such as FreeBSD, have borrowed code from Linux in return. Examples of this sharing include the Intel floating-point-emulation math library and the PC sound-hardware device drivers.

The Linux system as a whole is maintained by a loose network of developers collaborating over the Internet, with small groups or individuals having responsibility for maintaining the integrity of specific components. A small number of public Internet file-transfer-protocol (FTP) archive sites act as de facto standard repositories for these components. The **File System Hierarchy**

**Standard** document is also maintained by the Linux community as a means of ensuring compatibility across the various system components. This standard specifies the overall layout of a standard Linux file system; it determines under which directory names configuration files, libraries, system binaries, and run-time data files should be stored.

### 20.1.3 Linux Distributions

In theory, anybody can install a Linux system by fetching the latest revisions of the necessary system components from the `ftp` sites and compiling them. In Linux's early days, this is precisely what a Linux user had to do. As Linux has matured, however, various individuals and groups have attempted to make this job less painful by providing standard, precompiled sets of packages for easy installation.

These collections, or distributions, include much more than just the basic Linux system. They typically include extra system-installation and management utilities, as well as precompiled and ready-to-install packages of many of the common UNIX tools, such as news servers, web browsers, text-processing and editing tools, and even games.

The first distributions managed these packages by simply providing a means of unpacking all the files into the appropriate places. One of the important contributions of modern distributions, however, is advanced package management. Today's Linux distributions include a package-tracking database that allows packages to be installed, upgraded, or removed painlessly.

The SLS distribution, dating back to the early days of Linux, was the first collection of Linux packages that was recognizable as a complete distribution. Although it could be installed as a single entity, SLS lacked the package-management tools now expected of Linux distributions. The **Slackware** distribution represented a great improvement in overall quality, even though it also had poor package management. In fact, it is still one of the most widely installed distributions in the Linux community.

Since Slackware's release, many commercial and noncommercial Linux distributions have become available. **Red Hat** and **Debian** are particularly popular distributions; the first comes from a commercial Linux support company and the second from the free-software Linux community. Other commercially supported versions of Linux include distributions from **Canonical** and **SuSE**, and many others. There are too many Linux distributions in circulation for us to list all of them here. The variety of distributions does not prevent Linux distributions from being compatible, however. The RPM package file format is used, or at least understood, by the majority of distributions, and commercial applications distributed in this format can be installed and run on any distribution that can accept RPM files.

### 20.1.4 Linux Licensing

The Linux kernel is distributed under version 2.0 of the GNU General Public License (GPL), the terms of which are set out by the Free Software Foundation. Linux is not public-domain software. **Public domain** implies that the authors have waived copyright rights in the software, but copyright rights in Linux code are still held by the code's various authors. Linux is *free* software, however, in the sense that people can copy it, modify it, use it in any manner they want, and give away (or sell) their own copies.

The main implication of Linux's licensing terms is that nobody using Linux, or creating a derivative of Linux (a legitimate exercise), can distribute the derivative without including the source code. Software released under the GPL cannot be redistributed as a binary-only product. If you release software that includes any components covered by the GPL, then, under the GPL, you must make source code available alongside any binary distributions. (This restriction does not prohibit making—or even selling—binary software distributions, as long as anybody who receives binaries is also given the opportunity to get the originating source code for a reasonable distribution charge.)

## 20.2 Design Principles

In its overall design, Linux resembles other traditional, nonmicrokernel UNIX implementations. It is a multiuser, preemptively multitasking system with a full set of UNIX-compatible tools. Linux's file system adheres to traditional UNIX semantics, and the standard UNIX networking model is fully implemented. The internal details of Linux's design have been influenced heavily by the history of this operating system's development.

Although Linux runs on a wide variety of platforms, it was originally developed exclusively on PC architecture. A great deal of that early development was carried out by individual enthusiasts rather than by well-funded development or research facilities, so from the start Linux attempted to squeeze as much functionality as possible from limited resources. Today, Linux can run happily on a multiprocessor machine with hundreds of gigabytes of main memory and many terabytes of disk space, but it is still capable of operating usefully in under 16-MB of RAM.

As PCs became more powerful and as memory and hard disks became cheaper, the original, minimalist Linux kernels grew to implement more UNIX functionality. Speed and efficiency are still important design goals, but much recent and current work on Linux has concentrated on a third major design goal: standardization. One of the prices paid for the diversity of UNIX implementations currently available is that source code written for one may not necessarily compile or run correctly on another. Even when the same system calls are present on two different UNIX systems, they do not necessarily behave in exactly the same way. The POSIX standards comprise a set of specifications for different aspects of operating-system behavior. There are POSIX documents for common operating-system functionality and for extensions such as process threads and real-time operations. Linux is designed to comply with the relevant POSIX documents, and at least two Linux distributions have achieved official POSIX certification.

Because it gives standard interfaces to both the programmer and the user, Linux presents few surprises to anybody familiar with UNIX. We do not detail these interfaces here. The sections on the programmer interface (Section C.3) and user interface (Section C.4) of BSD apply equally well to Linux. By default, however, the Linux programming interface adheres to SVR4 UNIX semantics, rather than to BSD behavior. A separate set of libraries is available to implement BSD semantics in places where the two behaviors differ significantly.

Many other standards exist in the UNIX world, but full certification of Linux with respect to these standards is sometimes slowed because certification is often available only for a fee, and the expense involved in certifying an operating system's compliance with most standards is substantial. However, supporting a wide base of applications is important for any operating system, so implementation of standards is a major goal for Linux development, even without formal certification. In addition to the basic POSIX standard, Linux currently supports the POSIX threading extensions—Pthreads—and a subset of the POSIX extensions for real-time process control.

### 20.2.1 Components of a Linux System

The Linux system is composed of three main bodies of code, in line with most traditional UNIX implementations:

1. **Kernel**. The kernel is responsible for maintaining all the important abstractions of the operating system, including such things as virtual memory and processes.
2. **System libraries**. The system libraries define a standard set of functions through which applications can interact with the kernel. These functions implement much of the operating-system functionality that does not need the full privileges of kernel code. The most important system library is the **C library**, known as `libc`. In addition to providing the standard C library, `libc` implements the user mode side of the Linux system call interface, as well as other critical system-level interfaces.
3. **System utilities**. The system utilities are programs that perform individual, specialized management tasks. Some system utilities are invoked just once to initialize and configure some aspect of the system. Others—known as **daemons** in UNIX terminology—run permanently, handling such tasks as responding to incoming network connections, accepting logon requests from terminals, and updating log files.

Figure 20.1 illustrates the various components that make up a full Linux system. The most important distinction here is between the kernel and everything else. All the kernel code executes in the processor's privileged mode

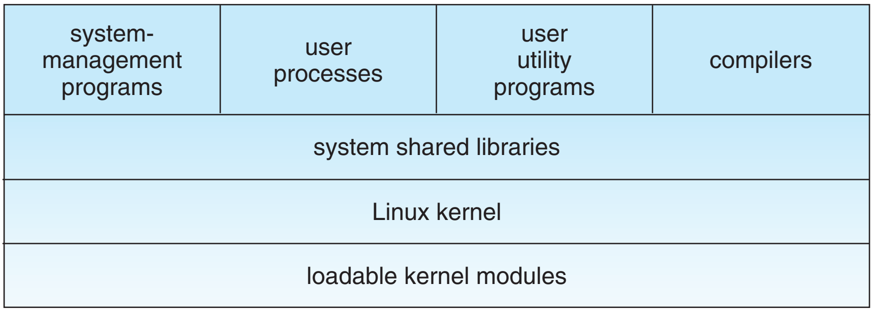


**Figure 20.1** Components of the Linux system.

with full access to all the physical resources of the computer. Linux refers to this privileged mode as **kernel mode**. Under Linux, no user code is built into the kernel. Any operating-system-support code that does not need to run in kernel mode is placed into the system libraries and runs in **user mode**. Unlike kernel mode, user mode has access only to a controlled subset of the system's resources.

Although various modern operating systems have adopted a message-passing architecture for their kernel internals, Linux retains UNIX's historical model: the kernel is created as a single, monolithic binary. The main reason is performance. Because all kernel code and data structures are kept in a single address space, no context switches are necessary when a thread calls an operating-system function or when a hardware interrupt is delivered. Moreover, the kernel can pass data and make requests between various subsystems using relatively cheap C function invocation and not more complicated interprocess communication (IPC). This single address space contains not only the core scheduling and virtual memory code but all kernel code, including all device drivers, file systems, and networking code.

Even though all the kernel components share this same melting pot, there is still room for modularity. In the same way that user applications can load shared libraries at run time to pull in a needed piece of code, so the Linux kernel can load (and unload) modules dynamically at run time. The kernel does not need to know in advance which modules may be loaded—they are truly independent loadable components.

The Linux kernel forms the core of the Linux operating system. It provides all the functionality necessary to manage processes and run threads, and it provides system services to give arbitrated and protected access to hardware resources. The kernel implements all the features required to qualify as an operating system. On its own, however, the operating system provided by the Linux kernel is not a complete UNIX system. It lacks much of the functionality and behavior of UNIX, and the features that it does provide are not necessarily in the format in which a UNIX application expects them to appear. The operating-system interface visible to running applications is not maintained directly by the kernel. Rather, applications make calls to the system libraries, which in turn call the operating-system services as necessary.

The system libraries provide many types of functionality. At the simplest level, they allow applications to make system calls to the Linux kernel. Making a system call involves transferring control from unprivileged user mode to privileged kernel mode; the details of this transfer vary from architecture to architecture. The libraries take care of collecting the system-call arguments and, if necessary, arranging those arguments in the special form necessary to make the system call.

The libraries may also provide more complex versions of the basic system calls. For example, the C language's buffered file-handling functions are all implemented in the system libraries, providing more advanced control of file I/O than the basic kernel system calls. The libraries also provide routines that do not correspond to system calls at all, such as sorting algorithms, mathematical functions, and string-manipulation routines. All the functions necessary to support the running of UNIX or POSIX applications are implemented in the system libraries.

The Linux system includes a wide variety of user-mode programs—both system utilities and user utilities. The system utilities include all the programs necessary to initialize and then administer the system, such as those to set up networking interfaces and to add and remove users from the system. User utilities are also necessary to the basic operation of the system but do not require elevated privileges to run. They include simple file-management utilities such as those to copy files, create directories, and edit text files. One of the most important user utilities is the **shell**, the standard command-line interface on UNIX systems. Linux supports many shells; the most common is the **bourne-again shell** (`bash`).

## 20.3 Kernel Modules

The Linux kernel has the ability to load and unload arbitrary sections of kernel code on demand. These loadable kernel modules run in privileged kernel mode and as a consequence have full access to all the hardware capabilities of the machine on which they run. In theory, there is no restriction on what a kernel module is allowed to do. Among other things, a kernel module can implement a device driver, a file system, or a networking protocol.

Kernel modules are convenient for several reasons. Linux's source code is free, so anybody wanting to write kernel code is able to compile a modified kernel and to reboot into that new functionality. However, recompiling, relinking, and reloading the entire kernel is a cumbersome cycle to undertake when you are developing a new driver. If you use kernel modules, you do not have to make a new kernel to test a new driver—the driver can be compiled on its own and loaded into the already running kernel. Of course, once a new driver is written, it can be distributed as a module so that other users can benefit from it without having to rebuild their kernels.

This latter point has another implication. Because it is covered by the GPL license, the Linux kernel cannot be released with proprietary components added to it unless those new components are also released under the GPL and the source code for them is made available on demand. The kernel's module interface allows third parties to write and distribute, on their own terms, device drivers or file systems that could not be distributed under the GPL.

Kernel modules allow a Linux system to be set up with a standard minimal kernel, without any extra device drivers built in. Any device drivers that the user needs can be either loaded explicitly by the system at startup or loaded automatically by the system on demand and unloaded when not in use. For example, a mouse driver can be loaded when a USB mouse is plugged into the system and unloaded when the mouse is unplugged.

The module support under Linux has four components:

1. The **module-management system** allows modules to be loaded into memory and to communicate with the rest of the kernel.
2. The **module loader and unloader**, which are user-mode utilities, work with the module-management system to load a module into memory.

3. The **driver-registration system** allows modules to tell the rest of the kernel that a new driver has become available.
4. A **conflict-resolutio mechanism** allows different device drivers to reserve hardware resources and to protect those resources from accidental use by another driver.

### 20.3.1 Module Management

Loading a module requires more than just loading its binary contents into kernel memory. The system must also make sure that any references the module makes to kernel symbols or entry points are updated to point to the correct locations in the kernel's address space. Linux deals with this reference updating by splitting the job of module loading into two separate sections: the management of sections of module code in kernel memory and the handling of symbols that modules are allowed to reference.

Linux maintains an internal symbol table in the kernel. This symbol table does not contain the full set of symbols defined in the kernel during the latter's compilation; rather, a symbol must be explicitly exported. The set of exported symbols constitutes a well-defined interface by which a module can interact with the kernel.

Although exporting symbols from a kernel function requires an explicit request by the programmer, no special effort is needed to import those symbols into a module. A module writer just uses the standard external linking of the C language. Any external symbols referenced by the module but not declared by it are simply marked as unresolved in the final module binary produced by the compiler. When a module is to be loaded into the kernel, a system utility first scans the module for these unresolved references. All symbols that still need to be resolved are looked up in the kernel's symbol table, and the correct addresses of those symbols in the currently running kernel are substituted into the module's code. Only then is the module passed to the kernel for loading. If the system utility cannot resolve all references in the module by looking them up in the kernel's symbol table, then the module is rejected.

The loading of the module is performed in two stages. First, the module-loader utility asks the kernel to reserve a continuous area of virtual kernel memory for the module. The kernel returns the address of the memory allocated, and the loader utility can use this address to relocate the module's machine code to the correct loading address. A second system call then passes the module, plus any symbol table that the new module wants to export, to the kernel. The module itself is now copied verbatim into the previously allocated space, and the kernel's symbol table is updated with the new symbols for possible use by other modules not yet loaded.

The final module-management component is the module requester. The kernel defines a communication interface to which a module-management program can connect. With this connection established, the kernel will inform the management process whenever a process requests a device driver, file system, or network service that is not currently loaded and will give the manager the opportunity to load that service. The original service request will complete once the module is loaded. The manager process regularly queries the kernel to see whether a dynamically loaded module is still in use and unloads that module when it is no longer actively needed.

### 20.3.2 Driver Registration

Once a module is loaded, it remains no more than an isolated region of memory until it lets the rest of the kernel know what new functionality it provides. The kernel maintains dynamic tables of all known drivers and provides a set of routines to allow drivers to be added to or removed from these tables at any time. The kernel makes sure that it calls a module's startup routine when that module is loaded and calls the module's cleanup routine before that module is unloaded. These routines are responsible for registering the module's functionality.

A module may register many types of functionality; it is not limited to only one type. For example, a device driver might want to register two separate mechanisms for accessing the device. Registration tables include, among others, the following items:

- **Device drivers**. These drivers include character devices (such as printers, terminals, and mice), block devices (including all disk drives), and network interface devices.
- **File systems**. The file system may be anything that implements Linux's virtual file system calling routines. It might implement a format for storing files on a disk, but it might equally well be a network file system, such as NFS, or a virtual file system whose contents are generated on demand, such as Linux's `/proc` file system.
- **Network protocols**. A module may implement an entire networking protocol, such as TCP, or simply a new set of packet-filtering rules for a network firewall.
- **Binary format**. This format specifies a way of recognizing, loading, and executing a new type of executable file.

In addition, a module can register a new set of entries in the `sysctl` and `/proc` tables, to allow that module to be configured dynamically (Section 20.7.4).

### 20.3.3 Conflict Resolution

Commercial UNIX implementations are usually sold to run on a vendor's own hardware. One advantage of a single-supplier solution is that the software vendor has a good idea about what hardware configurations are possible. PC hardware, however, comes in a vast number of configurations, with large numbers of possible drivers for devices such as network cards and video display adapters. The problem of managing the hardware configuration becomes more severe when modular device drivers are supported, since the currently active set of devices becomes dynamically variable.

Linux provides a central conflict-resolution mechanism to help arbitrate access to certain hardware resources. Its aims are as follows:

- To prevent modules from clashing over access to hardware resources
- To prevent **autoprobes**—device-driver probes that auto-detect device configuration—from interfering with existing device drivers

- To resolve conflicts among multiple drivers trying to access the same hardware—as, for example, when both the parallel printer driver and the parallel line IP (PLIP) network driver try to talk to the parallel port

To these ends, the kernel maintains lists of allocated hardware resources. The PC has a limited number of possible I/O ports (addresses in its hardware I/O address space), interrupt lines, and DMA channels. When any device driver wants to access such a resource, it is expected to reserve the resource with the kernel database first. This requirement incidentally allows the system administrator to determine exactly which resources have been allocated by which driver at any given point.

A module is expected to use this mechanism to reserve in advance any hardware resources that it expects to use. If the reservation is rejected because the resource is not present or is already in use, then it is up to the module to decide how to proceed. It may fail in its initialization attempt and request that it be unloaded if it cannot continue, or it may carry on, using alternative hardware resources.

## 20.4 Process Management

A process is the basic context in which all user-requested activity is serviced within the operating system. To be compatible with other UNIX systems, Linux must use a process model similar to those of other versions of UNIX. Linux operates differently from UNIX in a few key places, however. In this section, we review the traditional UNIX process model (Section C.3.2) and introduce Linux's threading model.

### 20.4.1 The fork() and exec() Process Model

The basic principle of UNIX process management is to separate into two steps two operations that are usually combined into one: the creation of a new process and the running of a new program. A new process is created by the `fork()` system call, and a new program is run after a call to `exec()`. These are two distinctly separate functions. We can create a new process with `fork()` without running a new program—the new subprocess simply continues to execute exactly the same program, at exactly the same point, that the first (parent) process was running. In the same way, running a new program does not require that a new process be created first. Any process may call `exec()` at any time. A new binary object is loaded into the process's address space and the new executable starts executing in the context of the existing process.

This model has the advantage of great simplicity. It is not necessary to specify every detail of the environment of a new program in the system call that runs that program. The new program simply runs in its existing environment. If a parent process wishes to modify the environment in which a new program is to be run, it can fork and then, still running the original executable in a child process, make any system calls it requires to modify that child process before finally executing the new program.

Under UNIX, then, a process encompasses all the information that the operating system must maintain to track the context of a single execution of a single program. Under Linux, we can break down this context into a number of specific sections. Broadly, process properties fall into three groups: the process identity, environment, and context.

#### 20.4.1.1 Process Identity

A process identity consists mainly of the following items:

- **Process ID (PID)**. Each process has a unique identifier. The PID is used to specify the process to the operating system when an application makes a system call to signal, modify, or wait for the process. Additional identifiers associate the process with a process group (typically, a tree of processes `fork`ed by a single user command) and login session.
- **Credentials**. Each process must have an associated user ID and one or more group IDs (user groups are discussed in Section 13.4.2) that determine the rights of a process to access system resources and files.
- **Personality**. Process personalities are not traditionally found on UNIX systems, but under Linux each process has an associated personality identifier that can slightly modify the semantics of certain system calls. Personalities are primarily used by emulation libraries to request that system calls be compatible with certain varieties of UNIX.
- **Namespace**. Each process is associated with a specific view of the file-system hierarchy, called its **namespace**. Most processes share a common namespace and thus operate on a shared file-system hierarchy. Processes and their children can, however, have different namespaces, each with a unique file-system hierarchy—their own root directory and set of mounted file systems.

Most of these identifiers are under the limited control of the process itself. The process group and session identifiers can be changed if the process wants to start a new group or session. Its credentials can be changed, subject to appropriate security checks. However, the primary PID of a process is unchangeable and uniquely identifies that process until termination.

#### 20.4.1.2 Process Environment

A process's environment is inherited from its parent and is composed of two null-terminated vectors: the argument vector and the environment vector. The **argument vector** simply lists the command-line arguments used to invoke the running program; it conventionally starts with the name of the program itself. The **environment vector** is a list of "NAME=VALUE" pairs that associates named environment variables with arbitrary textual values. The environment is not held in kernel memory but is stored in the process's own user-mode address space as the first datum at the top of the process's stack.

The argument and environment vectors are not altered when a new process is created. The new child process will inherit the environment of its parent. However, a completely new environment is set up when a new program is invoked. On calling `exec()`, a process must supply the environment for the new program. The kernel passes these environment variables to the next program, replacing the process's current environment. The kernel otherwise leaves the environment and command-line vectors alone—their interpretation is left entirely to the user-mode libraries and applications.

The passing of environment variables from one process to the next and the inheriting of these variables by the children of a process provide flexible ways to pass information to components of the user-mode system software. Various important environment variables have conventional meanings to related parts of the system software. For example, the `TERM` variable is set up to name the type of terminal connected to a user's login session. Many programs use this variable to determine how to perform operations on the user's display, such as moving the cursor and scrolling a region of text. Programs with multilingual support use the `LANG` variable to determine the language in which to display system messages for programs that include multilingual support.

The environment-variable mechanism custom-tailors the operating system on a per-process basis. Users can choose their own languages or select their own editors independently of one another.

#### 20.4.1.3 Process Context

The process identity and environment properties are usually set up when a process is created and not changed until that process exits. A process may choose to change some aspects of its identity if it needs to do so, or it may alter its environment. In contrast, process context is the state of the running program at any one time; it changes constantly. Process context includes the following parts:

- **Scheduling context**. The most important part of the process context is its scheduling context—the information that the scheduler needs to suspend and restart the process. This information includes saved copies of all the process's registers. Floating-point registers are stored separately and are restored only when needed. Thus, processes that do not use floating-point arithmetic do not incur the overhead of saving that state. The scheduling context also includes information about scheduling priority and about any outstanding signals waiting to be delivered to the process. A key part of the scheduling context is the process's kernel stack, a separate area of kernel memory reserved for use by kernel-mode code. Both system calls and interrupts that occur while the process is executing will use this stack.
- **Accounting**. The kernel maintains accounting information about the resources currently being consumed by each process and the total resources consumed by the process in its entire lifetime so far.
- **File table**. The file table is an array of pointers to kernel file structures representing open files. When making file-I/O system calls, processes refer to files by an integer, known as a **file descriptor** (**fd**), that the kernel uses to index into this table.
- **File-system context**. Whereas the file table lists the existing open files, the file-system context applies to requests to open new files. The file-system context includes the process's root directory, current working directory, and namespace.

- **Signal-handler table**. UNIX systems can deliver asynchronous signals to a process in response to various external events. The signal-handler table defines the action to take in response to a specific signal. Valid actions include ignoring the signal, terminating the process, and invoking a routine in the process's address space.
- **Virtual memory context**. The virtual memory context describes the full contents of a process's private address space; we discuss it in Section 20.6.

### 20.4.2 Processes and Threads

Linux provides the `fork()` system call, which duplicates a process without loading a new executable image. Linux also provides the ability to create threads via the `clone()` system call. Linux does not distinguish between processes and threads, however. In fact, Linux generally uses the term ***task***—rather than ***process*** or ***thread***—when referring to a flow of control within a program. The `clone()` system call behaves identically to `fork()`, except that it accepts as arguments a set of flags that dictate what resources are shared between the parent and child (whereas a process created with `fork()` shares no resources with its parent). The flags include:

| flag | meaning |
|---|---|
| `CLONE_FS` | File-system information is shared. |
| `CLONE_VM` | The same memory space is shared. |
| `CLONE_SIGHAND` | Signal handlers are shared. |
| `CLONE_FILES` | The set of open files is shared. |

Thus, if `clone()` is passed the flags `CLONE_FS`, `CLONE_VM`, `CLONE_SIGHAND`, and `CLONE_FILES`, the parent and child tasks will share the same file-system information (such as the current working directory), the same memory space, the same signal handlers, and the same set of open files. Using `clone()` in this fashion is equivalent to creating a thread in other systems, since the parent task shares most of its resources with its child task. If none of these flags is set when `clone()` is invoked, however, the associated resources are not shared, resulting in functionality similar to that of the `fork()` system call.

The lack of distinction between processes and threads is possible because Linux does not hold a process's entire context within the main process data structure. Rather, it holds the context within independent subcontexts. Thus, a process's file-system context, file-descriptor table, signal-handler table, and virtual memory context are held in separate data structures. The process data structure simply contains pointers to these other structures, so any number of processes can easily share a subcontext by pointing to the same subcontext and incrementing a reference count.

The arguments to the `clone()` system call tell it which subcontexts to copy and which to share. The new process is always given a new identity and a new scheduling context—these are the essentials of a Linux process. According to the arguments passed, however, the kernel may either create new subcontext data structures initialized so as to be copies of the parent's or set up the new process to use the same subcontext data structures being used by the parent.

The `fork()` system call is nothing more than a special case of `clone()` that copies all subcontexts, sharing none.

## 20.5 Scheduling

Scheduling is the job of allocating CPU time to different tasks within an operating system. Linux, like all UNIX systems, supports **preemptive multitasking**. In such a system, the process scheduler decides which thread runs and when. Making these decisions in a way that balances fairness and performance across many different workloads is one of the more complicated challenges in modern operating systems.

Normally, we think of scheduling as the running and interrupting of user threads, but another aspect of scheduling is also important in Linux: the running of the various kernel tasks. Kernel tasks encompass both tasks that are requested by a running thread and tasks that execute internally on behalf of the kernel itself, such as tasks spawned by Linux's I/O subsystem.

### 20.5.1 Thread Scheduling

Linux has two separate process-scheduling algorithms. One is a time-sharing algorithm for fair, preemptive scheduling among multiple threads. The other is designed for real-time tasks, where absolute priorities are more important than fairness.

The scheduling algorithm used for routine time-sharing tasks received a major overhaul with version 2.6 of the kernel. Earlier versions ran a variation of the traditional UNIX scheduling algorithm. This algorithm does not provide adequate support for SMP systems, does not scale well as the number of tasks on the system grows, and does not maintain fairness among interactive tasks, particularly on systems such as desktops and mobile devices. The thread scheduler was first overhauled with version 2.5 of the kernel. Version 2.5 implemented a scheduling algorithm that selects which task to run in constant time—known as $O(1)$—regardless of the number of tasks or processors in the system. The new scheduler also provided increased support for SMP, including processor affinity and load balancing. These changes, while improving scalability, did not improve interactive performance or fairness—and, in fact, made these problems worse under certain workloads. Consequently, the thread scheduler was overhauled a second time, with Linux kernel version 2.6. This version ushered in the **Completely Fair Scheduler** (**CFS**).

The Linux scheduler is a preemptive, priority-based algorithm with two separate priority ranges: a **real-time** range from 0 to 99 and a **nice value** ranging from −20 to 19. Smaller `nice` values indicate higher priorities. Thus, by increasing the `nice` value, you are decreasing your priority and being "nice" to the rest of the system.

CFS is a significant departure from the traditional UNIX process scheduler. In the latter, the core variables in the scheduling algorithm are priority and time slice. The **time slice** is the length of time—the ***slice*** of the processor —that a thread is afforded. Traditional UNIX systems give processes a fixed time slice, perhaps with a boost or penalty for high- or low-priority processes, respectively. A process may run for the length of its time slice, and higher-priority processes run before lower-priority processes. It is a simple algorithm that many non-UNIX systems employ. Such simplicity worked well for early time-sharing systems but has proved incapable of delivering good interactive performance and fairness on today's modern desktops and mobile devices.

CFS introduced a new scheduling algorithm called **fair scheduling** that eliminates time slices in the traditional sense. Instead of time slices, all threads are allotted a proportion of the processor's time. CFS calculates how long a thread should run as a function of the total number of runnable threads. To start, CFS says that if there are $N$ runnable threads, then each should be afforded $1/N$ of the processor's time. CFS then adjusts this allotment by weighting each thread's allotment by its `nice` value. Threads with the default `nice` value have a weight of 1—their priority is unchanged. Threads with a smaller `nice` value (higher priority) receive a higher weight, while threads with a larger `nice` value (lower priority) receive a lower weight. CFS then runs each thread for a "time slice" proportional to the process's weight divided by the total weight of all runnable processes.

To calculate the actual length of time a thread runs, CFS relies on a configurable variable called **target latency**, which is the interval of time during which every runnable task should run at least once. For example, assume that the target latency is 10 milliseconds. Further assume that we have two runnable threads of the same priority. Each of these threads has the same weight and therefore receives the same proportion of the processor's time. In this case, with a target latency of 10 milliseconds, the first process runs for 5 milliseconds, then the other process runs for 5 milliseconds, then the first process runs for 5 milliseconds again, and so forth. If we have 10 runnable threads, then CFS will run each for a millisecond before repeating.

But what if we had, say, 1,000 threads? Each thread would run for 1 microsecond if we followed the procedure just described. Due to switching costs, scheduling threads for such short lengths of time is inefficient. CFS consequently relies on a second configurable variable, the **minimum granularity**, which is a minimum length of time any thread is allotted the processor. All threads, regardless of the target latency, will run for at least the minimum granularity. In this manner, CFS ensures that switching costs do not grow unacceptably large when the number of runnable threads increases significantly. In doing so, it violates its attempts at fairness. In the usual case, however, the number of runnable threads remains reasonable, and both fairness and switching costs are maximized.

With the switch to fair scheduling, CFS behaves differently from traditional UNIX process schedulers in several ways. Most notably, as we have seen, CFS eliminates the concept of a static time slice. Instead, each thread receives a proportion of the processor's time. How long that allotment is depends on how many other threads are runnable. This approach solves several problems in mapping priorities to time slices inherent in preemptive, priority-based scheduling algorithms. It is possible, of course, to solve these problems in other ways without abandoning the classic UNIX scheduler. CFS, however, solves the problems with a simple algorithm that performs well on interactive workloads such as mobile devices without compromising throughput performance on the largest of servers.

### 20.5.2 Real-Time Scheduling

Linux's real-time scheduling algorithm is significantly simpler than the fair scheduling employed for standard time-sharing threads. Linux implements the two real-time scheduling classes required by POSIX.1b: first-come, first-served (FCFS) and round-robin (Section 5.3.1 and Section 5.3.3, respectively). In both cases, each thread has a priority in addition to its scheduling class. The scheduler always runs the thread with the highest priority. Among threads of equal priority, it runs the thread that has been waiting longest. The only difference between FCFS and round-robin scheduling is that FCFS threads continue to run until they either exit or block, whereas a round-robin thread will be preempted after a while and will be moved to the end of the scheduling queue, so round-robin threads of equal priority will automatically time-share among themselves.

Linux's real-time scheduling is soft—rather than hard—real time. The scheduler offers strict guarantees about the relative priorities of real-time threads, but the kernel does not offer any guarantees about how quickly a real-time thread will be scheduled once that thread becomes runnable. In contrast, a hard real-time system can guarantee a minimum latency between when a thread becomes runnable and when it actually runs.

### 20.5.3 Kernel Synchronization

The way the kernel schedules its own operations is fundamentally different from the way it schedules threads. A request for kernel-mode execution can occur in two ways. A running program may request an operating-system service, either explicitly via a system call or implicitly—for example, when a page fault occurs. Alternatively, a device controller may deliver a hardware interrupt that causes the CPU to start executing a kernel-defined handler for that interrupt.

The problem for the kernel is that all these tasks may try to access the same internal data structures. If one kernel task is in the middle of accessing some data structure when an interrupt service routine executes, then that service routine cannot access or modify the same data without risking data corruption. This fact relates to the idea of critical sections—portions of code that access shared data and thus must not be allowed to execute concurrently. As a result, kernel synchronization involves much more than just thread scheduling. A framework is required that allows kernel tasks to run without violating the integrity of shared data.

Prior to version 2.6, Linux was a nonpreemptive kernel, meaning that a thread running in kernel mode could not be preempted—even if a higher-priority thread became available to run. With version 2.6, the Linux kernel became fully preemptive. Now, a task can be preempted when it is running in the kernel.

The Linux kernel provides spinlocks and semaphores (as well as reader–writer versions of these two locks) for locking in the kernel. On SMP machines, the fundamental locking mechanism is a spinlock, and the kernel is designed so that spinlocks are held for only short durations. On single-processor machines, spinlocks are not appropriate for use and are replaced by enabling and disabling kernel preemption. That is, rather than holding a spinlock, the task disables kernel preemption. When the task would otherwise release the spinlock, it enables kernel preemption. This pattern is summarized below:

| single processor | multiple processors |
|---|---|
| Disable kernel preemption. | Acquire spin lock. |
| Enable kernel preemption. | Release spin lock. |

Linux uses an interesting approach to disable and enable kernel preemption. It provides two simple kernel interfaces—`preempt_disable()` and `preempt_enable()`. In addition, the kernel is not preemptible if a kernel-mode task is holding a spinlock. To enforce this rule, each task in the system has a `thread-info` structure that includes the field `preempt_count`, which is a counter indicating the number of locks being held by the task. The counter is incremented when a lock is acquired and decremented when a lock is released. If the value of `preempt_count` for the task currently running is greater than zero, it is not safe to preempt the kernel, as this task currently holds a lock. If the count is zero, the kernel can safely be interrupted, assuming there are no outstanding calls to `preempt_disable()`.

Spinlocks—along with the enabling and disabling of kernel preemption—are used in the kernel only when the lock is held for short durations. When a lock must be held for longer periods, semaphores are used.

The second protection technique used by Linux applies to critical sections that occur in interrupt service routines. The basic tool is the processor's interrupt-control hardware. By disabling interrupts (or using spinlocks) during a critical section, the kernel guarantees that it can proceed without the risk of concurrent access to shared data structures.

However, there is a penalty for disabling interrupts. On most hardware architectures, interrupt enable and disable instructions are not cheap. More importantly, as long as interrupts remain disabled, all I/O is suspended, and any device waiting for servicing will have to wait until interrupts are reenabled; thus, performance degrades. To address this problem, the Linux kernel uses a synchronization architecture that allows long critical sections to run for their entire duration without having interrupts disabled. This ability is especially useful in the networking code. An interrupt in a network device driver can signal the arrival of an entire network packet, which may result in a great deal of code being executed to disassemble, route, and forward that packet within the interrupt service routine.

Linux implements this architecture by separating interrupt service routines into two sections: the top half and the bottom half. The ***top half*** is the standard interrupt service routine that runs with recursive interrupts disabled. Interrupts of the same number (or line) are disabled, but other interrupts may run. The ***bottom half*** of a service routine is run, with all interrupts enabled, by a miniature scheduler that ensures that bottom halves never interrupt themselves. The bottom-half scheduler is invoked automatically whenever an interrupt service routine exits.

This separation means that the kernel can complete any complex processing that has to be done in response to an interrupt without worrying about being interrupted itself. If another interrupt occurs while a bottom half is executing, then that interrupt can request that the same bottom half execute, but the execution will be deferred until the one currently running completes. Each execution of the bottom half can be interrupted by a top half but can never be interrupted by a similar bottom half.

The top-half/bottom-half architecture is completed by a mechanism for disabling selected bottom halves while executing normal, foreground kernel code. The kernel can code critical sections easily using this system. Interrupt handlers can code their critical sections as bottom halves; and when the foreground kernel wants to enter a critical section, it can disable any relevant bottom halves to prevent any other critical sections from interrupting it. At the end of the critical section, the kernel can reenable the bottom halves and run any bottom-half tasks that have been queued by top-half interrupt service routines during the critical section.

Figure 20.2 summarizes the various levels of interrupt protection within the kernel. Each level may be interrupted by code running at a higher level but will never be interrupted by code running at the same or a lower level. Except for user-mode code, user threads can always be preempted by another thread when a time-sharing scheduling interrupt occurs.

### 20.5.4 Symmetric Multiprocessing

The Linux 2.0 kernel was the first stable Linux kernel to support **symmetric multiprocessor** (**SMP**) hardware, allowing separate threads to execute in parallel on separate processors. The original implementation of SMP imposed the restriction that only one processor at a time could be executing kernel code.

In version 2.2 of the kernel, a single kernel spinlock (sometimes termed **BKL** for “big kernel lock”) was created to allow multiple threads (running on different processors) to be active in the kernel concurrently. However, the BKL provided a very coarse level of locking granularity, resulting in poor scalability to machines with many processors and threads. Later releases of the kernel made the SMP implementation more scalable by splitting this single kernel spinlock into multiple locks, each of which protects only a small subset of the kernel’s data structures. Such spinlocks were described in Section 20.5.3.

The 3.0 and 4.0 kernels provided additional SMP enhancements, including ever-finer locking, processor affinity, load-balancing algorithms, and support for hundreds or even thousands of physical processors in a single system.

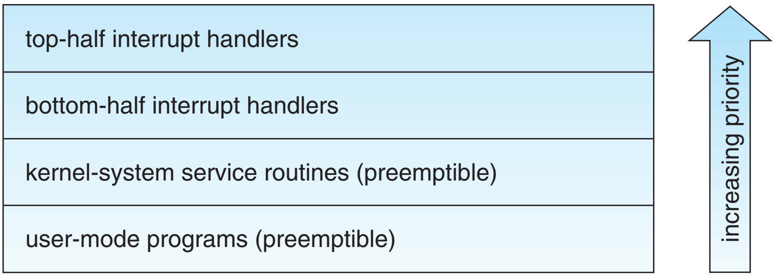

top-half interrupt handlers
bottom-half interrupt handlers
kernel-system service routines (preemptible)
user-mode programs (preemptible)
increasing priority

**Figure 20.2** Interrupt protection levels.

## 20.6 Memory Management

Memory management under Linux has two components. The first deals with allocating and freeing physical memory—pages, groups of pages, and small blocks of RAM. The second handles virtual memory, which is memory-mapped into the address space of running processes. In this section, we describe these two components and then examine the mechanisms by which the loadable components of a new program are brought into a process's virtual memory in response to an `exec()` system call.

### 20.6.1 Management of Physical Memory

Due to specific hardware constraints, Linux separates physical memory into four different **zones**, or regions:

- `ZONE_DMA`
- `ZONE_DMA32`
- `ZONE_NORMAL`
- `ZONE_HIGHMEM`

These zones are architecture specific. For example, on the Intel x86-32 architecture, certain ISA (industry standard architecture) devices can only access the lower 16-MB of physical memory using DMA. On these systems, the first 16-MB of physical memory comprise `ZONE_DMA`. On other systems, certain devices can only access the first 4-GB of physical memory, despite supporting 64-bit addresses. On such systems, the first 4 GB of physical memory comprise `ZONE_DMA32`. `ZONE_HIGHMEM` (for "high memory") refers to physical memory that is not mapped into the kernel address space. For example, on the 32-bit Intel architecture (where $2^{32}$ provides a 4-GB address space), the kernel is mapped into the first 896 MB of the address space; the remaining memory is referred to as ***high memory*** and is allocated from `ZONE_HIGHMEM`. Finally, `ZONE_NORMAL` comprises everything else—the normal, regularly mapped pages. Whether an architecture has a given zone depends on its constraints. A modern, 64-bit architecture such as Intel x86-64 has a small 16-MB `ZONE_DMA` (for legacy devices) and all the rest of its memory in `ZONE_NORMAL`, with no "high memory".

The relationship of zones and physical addresses on the Intel x86-32 architecture is shown in Figure 20.3. The kernel maintains a list of free pages for

| zone | physical memory |
|---|---|
| ZONE_DMA | < 16 MB |
| ZONE_NORMAL | 16 .. 896 MB |
| ZONE_HIGHMEM | > 896 MB |

**Figure 20.3** Relationship of zones and physical addresses in Intel x86-32.

each zone. When a request for physical memory arrives, the kernel satisfies the request using the appropriate zone.

The primary physical-memory manager in the Linux kernel is the **page allocator**. Each zone has its own allocator, which is responsible for allocating and freeing all physical pages for the zone and is capable of allocating ranges of physically contiguous pages on request. The allocator uses a buddy system (Section 10.8.1) to keep track of available physical pages. In this scheme, adjacent units of allocatable memory are paired together (hence its name). Each allocatable memory region has an adjacent partner or buddy. Whenever two allocated partner regions are freed up, they are combined to form a larger region—a **buddy heap**. That larger region also has a partner, with which it can combine to form a still larger free region. Conversely, if a small memory request cannot be satisfied by allocation of an existing small free region, then a larger free region will be subdivided into two partners to satisfy the request. Separate linked lists are used to record the free memory regions of each allowable size. Under Linux, the smallest size allocatable under this mechanism is a single physical page. Figure 20.4 shows an example of buddy-heap allocation. A 4-KB region is being allocated, but the smallest available region is 16 KB. The region is broken up recursively until a piece of the desired size is available.

Ultimately, all memory allocations in the Linux kernel are made either statically, by drivers that reserve a contiguous area of memory during system boot time, or dynamically, by the page allocator. However, kernel functions do not have to use the basic allocator to reserve memory. Several specialized memory-management subsystems use the underlying page allocator to manage their own pools of memory. The most important are the virtual memory system, described in Section 20.6.2; the `kmalloc()` variable-length allocator; the slab allocator, used for allocating memory for kernel data structures; and the page cache, used for caching pages belonging to files.

Many components of the Linux operating system need to allocate entire pages on request, but often smaller blocks of memory are required. The kernel provides an additional allocator for arbitrary-sized requests, where the size of a request is not known in advance and may be only a few bytes. Analogous to the C language's `malloc()` function, this `kmalloc()` service allocates entire physical pages on demand but then splits them into smaller pieces. The

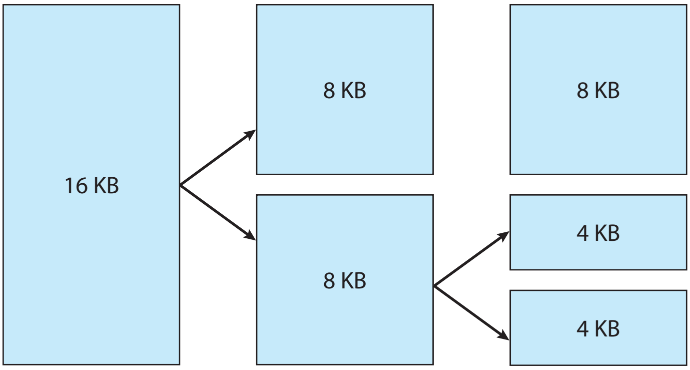


**Figure 20.4** Splitting of memory in the buddy system.

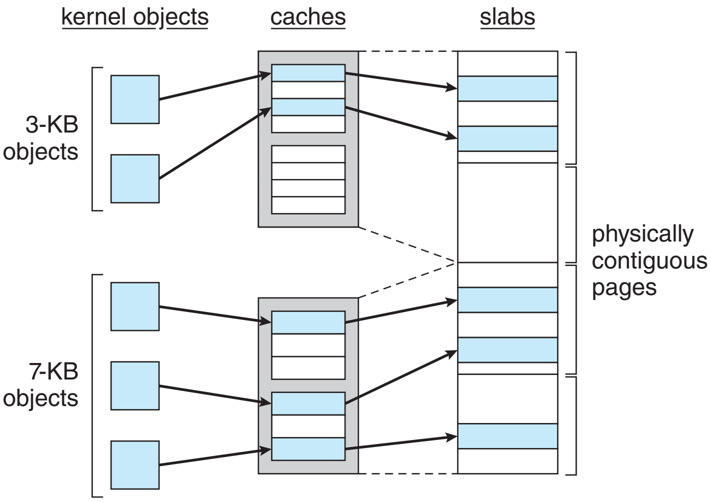


**Figure 20.5** Slab allocator in Linux.

kernel maintains lists of pages in use by the `kmalloc()` service. Allocating memory involves determining the appropriate list and either taking the first free piece available on the list or allocating a new page and splitting it up. Memory regions claimed by the `kmalloc()` system are allocated permanently until they are freed explicitly with a corresponding call to `kfree()`; the `kmalloc()` system cannot reallocate or reclaim these regions in response to memory shortages.

Another strategy adopted by Linux for allocating kernel memory is known as slab allocation. A **slab** is used for allocating memory for kernel data structures and is made up of one or more physically contiguous pages. A **cache** consists of one or more slabs. There is a single cache for each unique kernel data structure—for example, a cache for the data structure representing process descriptors, a cache for file objects, a cache for inodes, and so forth. Each cache is populated with **objects** that are instantiations of the kernel data structure the cache represents. For example, the cache representing inodes stores instances of inode structures, and the cache representing process descriptors stores instances of process descriptor structures. The relationship among slabs, caches, and objects is shown in Figure 20.5. The figure shows two kernel objects 3 KB in size and three objects 7 KB in size. These objects are stored in the respective caches for 3-KB and 7-KB objects.

The slab-allocation algorithm uses caches to store kernel objects. When a cache is created, a number of objects are allocated to the cache. The number of objects in the cache depends on the size of the associated slab. For example, a 12-KB slab (made up of three contiguous 4-KB pages) could store six 2-KB objects. Initially, all the objects in the cache are marked as `free`. When a new object for a kernel data structure is needed, the allocator can assign any free object from the cache to satisfy the request. The object assigned from the cache is marked as `used`.

Let's consider a scenario in which the kernel requests memory from the slab allocator for an object representing a process descriptor. In Linux systems, a process descriptor is of the type `struct task_struct`, which requires approximately 1.7 KB of memory. When the Linux kernel creates a new task, it requests the necessary memory for the `struct task_struct` object from its cache. The cache will fulfill the request using a `struct task_struct` object that has already been allocated in a slab and is marked as free.

In Linux, a slab may be in one of three possible states:

1. **Full**. All objects in the slab are marked as used.
2. **Empty**. All objects in the slab are marked as free.
3. **Partial**. The slab consists of both used and free objects.

The slab allocator first attempts to satisfy the request with a free object in a partial slab. If none exists, a free object is assigned from an empty slab. If no empty slabs are available, a new slab is allocated from contiguous physical pages and assigned to a cache; memory for the object is allocated from this slab.

Two other main subsystems in Linux do their own management of physical pages: the page cache and the virtual memory system. These systems are closely related to each other. The **page cache** is the kernel's main cache for files and is the main mechanism through which I/O to block devices (Section 20.8.1) is performed. File systems of all types, including the native Linux disk-based file systems and the NFS networked file system, perform their I/O through the page cache. The page cache stores entire pages of file contents and is not limited to block devices. It can also cache networked data. The virtual memory system manages the contents of each process's virtual address space. These two systems interact closely with each other because reading a page of data into the page cache requires mapping pages in the page cache using the virtual memory system. In the following section, we look at the virtual memory system in greater detail.

### 20.6.2 Virtual Memory

The Linux virtual memory system is responsible for maintaining the address space accessible to each process. It creates pages of virtual memory on demand and manages loading those pages from disk and swapping them back out to disk as required. Under Linux, the virtual memory manager maintains two separate views of a process's address space: as a set of separate regions and as a set of pages.

The first view of an address space is the logical view, describing instructions that the virtual memory system has received concerning the layout of the address space. In this view, the address space consists of a set of nonoverlapping regions, each region representing a continuous, page-aligned subset of the address space. Each region is described internally by a single `vm_area_struct` structure that defines the properties of the region, including the process's read, write, and execute permissions in the region as well as information about any files associated with the region. The regions for each address space are linked into a balanced binary tree to allow fast lookup of the region corresponding to any virtual address.

The kernel also maintains a second, physical view of each address space. This view is stored in the hardware page tables for the process. The page-table entries identify the exact current location of each page of virtual memory, whether it is on disk or in physical memory. The physical view is managed by a set of routines, which are invoked from the kernel's software-interrupt handlers whenever a process tries to access a page that is not currently present in the page tables. Each `vm_area_struct` in the address-space description contains a field pointing to a table of functions that implement the key page-management functionality for any given virtual memory region. All requests to read or write an unavailable page are eventually dispatched to the appropriate handler in the function table for the `vm_area_struct`, so that the central memory-management routines do not have to know the details of managing each possible type of memory region.

#### 20.6.2.1 Virtual Memory Regions

Linux implements several types of virtual memory regions. One property that characterizes virtual memory is the backing store for the region, which describes where the pages for the region come from. Most memory regions are backed either by a file or by nothing. A region backed by nothing is the simplest type of virtual memory region. Such a region represents **demand-zero memory**: when a process tries to read a page in such a region, it is simply given back a page of memory filled with zeros.

A region backed by a file acts as a viewport onto a section of that file. Whenever the process tries to access a page within that region, the page table is filled with the address of a page within the kernel's page cache corresponding to the appropriate offset in the file. The same page of physical memory is used by both the page cache and the process's page tables, so any changes made to the file by the file system are immediately visible to any processes that have mapped that file into their address space. Any number of processes can map the same region of the same file, and they will all end up using the same page of physical memory for the purpose.

A virtual memory region is also defined by its reaction to writes. The mapping of a region into the process's address space can be either ***private*** or ***shared***. If a process writes to a privately mapped region, then the pager detects that a copy-on-write is necessary to keep the changes local to the process. In contrast, writes to a shared region result in updating of the object mapped into that region, so that the change will be visible immediately to any other process that is mapping that object.

#### 20.6.2.2 Lifetime of a Virtual Address Space

The kernel creates a new virtual address space in two situations: when a process runs a new program with the `exec()` system call and when a new process is created by the `fork()` system call. The first case is easy. When a new program is executed, the process is given a new, completely empty virtual address space. It is up to the routines for loading the program to populate the address space with virtual memory regions.

The second case, creating a new process with `fork()`, involves creating a complete copy of the existing process's virtual address space. The kernel copies the parent process's `vm_area_struct` descriptors, then creates a new set of page tables for the child. The parent's page tables are copied directly into the child's, and the reference count of each page covered is incremented. Thus, after the fork, the parent and child share the same physical pages of memory in their address spaces.

A special case occurs when the copying operation reaches a virtual memory region that is mapped privately. Any pages to which the parent process has written within such a region are private, and subsequent changes to these pages by either the parent or the child must not update the page in the other process's address space. When the page-table entries for such regions are copied, they are set to be read only and are marked for copy-on-write. As long as neither process modifies these pages, the two processes share the same page of physical memory. However, if either process tries to modify a copy-on-write page, the reference count on the page is checked. If the page is still shared, then the process copies the page's contents to a brand-new page of physical memory and uses its copy instead. This mechanism ensures that private data pages are shared between processes whenever possible and copies are made only when absolutely necessary.

#### 20.6.2.3 Swapping and Paging

An important task for a virtual memory system is to relocate pages of memory from physical memory out to disk when that memory is needed. Early UNIX systems performed this relocation by swapping out the contents of entire processes at once, but modern versions of UNIX rely more on paging—the movement of individual pages of virtual memory between physical memory and disk. Linux does not implement whole-process swapping; it uses the newer paging mechanism exclusively.

The paging system can be divided into two sections. First, the **policy algorithm** decides which pages to write out to backing store and when to write them. Second, the **paging mechanism** carries out the transfer and pages data back into physical memory when they are needed again.

Linux's **pageout policy** uses a modified version of the standard clock (or second-chance) algorithm described in Section 10.4.5.2. Under Linux, a multiple-pass clock is used, and every page has an ***age*** that is adjusted on each pass of the clock. The age is more precisely a measure of the page's youthfulness, or how much activity the page has seen recently. Frequently accessed pages will attain a higher age value, but the age of infrequently accessed pages will drop toward zero with each pass. This age valuing allows the pager to select pages to page out based on a least frequently used (LFU) policy.

The paging mechanism supports paging both to dedicated swap devices and partitions and to normal files, although swapping to a file is significantly slower due to the extra overhead incurred by the file system. Blocks are allocated from the swap devices according to a bitmap of used blocks, which is maintained in physical memory at all times. The allocator uses a next-fit algorithm to try to write out pages to continuous runs of secondary storage blocks for improved performance. The allocator records the fact that a page has been paged out to storage by using a feature of the page tables on modern processors: the page-table entry's page-not-present bit is set, allowing the rest of the page-table entry to be filled with an index identifying where the page has been written.

#### 20.6.2.4 Kernel Virtual Memory

Linux reserves for its own internal use a constant, architecture-dependent region of the virtual address space of every process. The page-table entries that map to these kernel pages are marked as protected, so that the pages are not visible or modifiable when the processor is running in user mode. This kernel virtual memory area contains two regions. The first is a static area that contains page-table references to every available physical page of memory in the system, so that a simple translation from physical to virtual addresses occurs when kernel code is run. The core of the kernel, along with all pages allocated by the normal page allocator, resides in this region.

The remainder of the kernel's reserved section of address space is not reserved for any specific purpose. Page-table entries in this address range can be modified by the kernel to point to any other areas of memory. The kernel provides a pair of facilities that allow kernel code to use this virtual memory. The `vmalloc()` function allocates an arbitrary number of physical pages of memory that may not be physically contiguous into a single region of virtually contiguous kernel memory. The `vremap()` function maps a sequence of virtual addresses to point to an area of memory used by a device driver for memory-mapped I/O.

### 20.6.3 Execution and Loading of User Programs

The Linux kernel's execution of user programs is triggered by a call to the `exec()` system call. This `exec()` call commands the kernel to run a new program within the current process, completely overwriting the current execution context with the initial context of the new program. The first job of this system service is to verify that the calling process has permission rights to the file being executed. Once that matter has been checked, the kernel invokes a loader routine to start running the program. The loader does not necessarily load the contents of the program file into physical memory, but it does at least set up the mapping of the program into virtual memory.

There is no single routine in Linux for loading a new program. Instead, Linux maintains a table of possible loader functions, and it gives each such function the opportunity to try loading the given file when an `exec()` system call is made. The initial reason for this loader table was that, between the releases of the 1.0 and 1.2 kernels, the standard format for Linux's binary files was changed. Older Linux kernels understood the `a.out` format for binary files—a relatively simple format common on older UNIX systems. Newer Linux systems use the more modern **ELF** format, now supported by most current UNIX implementations. ELF has a number of advantages over `a.out`, including flexibility and extendability. New sections can be added to an ELF binary (for example, to add extra debugging information) without causing the loader routines to become confused. By allowing registration of multiple loader routines, Linux can easily support the ELF and `a.out` binary formats in a single running system.

In Section 20.6.3.1 and Section 20.6.3.2, we concentrate exclusively on the loading and running of ELF-format binaries. The procedure for loading `a.out` binaries is simpler but similar in operation.

#### 20.6.3.1 Mapping of Programs into Memory

Under Linux, the binary loader does not load a binary file into physical memory. Rather, the pages of the binary file are mapped into regions of virtual memory. Only when the program tries to access a given page will a page fault result in the loading of that page into physical memory using demand paging.

It is the responsibility of the kernel's binary loader to set up the initial memory mapping. An ELF-format binary file consists of a header followed by several page-aligned sections. The ELF loader works by reading the header and mapping the sections of the file into separate regions of virtual memory.

Figure 20.6 shows the typical layout of memory regions set up by the ELF loader. In a reserved region at one end of the address space sits the kernel, in its own privileged region of virtual memory inaccessible to normal user-mode programs. The rest of virtual memory is available to applications, which can use the kernel's memory-mapping functions to create regions that map a portion of a file or that are available for application data.

The loader's job is to set up the initial memory mapping to allow the execution of the program to start. The regions that need to be initialized include the stack and the program's text and data regions.

The stack is created at the top of the user-mode virtual memory; it grows downward toward lower-numbered addresses. It includes copies of the arguments and environment variables given to the program in the `exec()` system call. The other regions are created near the bottom end of virtual memory. The sections of the binary file that contain program text or read-only data are mapped into memory as a write-protected region. Writable initialized data are mapped next; then any uninitialized data are mapped in as a private demand-zero region.

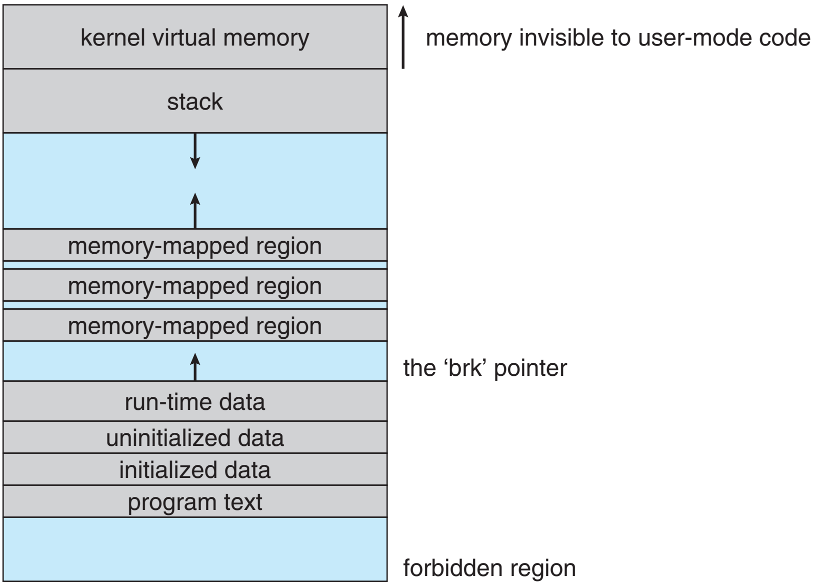

kernel virtual memory
memory invisible to user-mode code
stack
memory-mapped region
memory-mapped region
memory-mapped region
the 'brk' pointer
run-time data
uninitialized data
initialized data
program text
forbidden region

**Figure 20.6** Memory layout for ELF programs.

Directly beyond these fixed-sized regions is a variable-sized region that programs can expand as needed to hold data allocated at run time. Each process has a pointer, `brk`, that points to the current extent of this data region, and processes can extend or contract their `brk` region with a single system call—`sbrk()`.

Once these mappings have been set up, the loader initializes the process's program-counter register with the starting point recorded in the ELF header, and the process can be scheduled.

#### 20.6.3.2 Static and Dynamic Linking

Once the program has been loaded and has started running, all the necessary contents of the binary file have been loaded into the process's virtual address space. However, most programs also need to run functions from the system libraries, and these library functions must also be loaded. In the simplest case, the necessary library functions are embedded directly in the program's executable binary file. Such a program is statically linked to its libraries, and statically linked executables can commence running as soon as they are loaded.

The main disadvantage of static linking is that every program generated must contain copies of exactly the same common system library functions. It is much more efficient, in terms of both physical memory and disk-space usage, to load the system libraries into memory only once. Dynamic linking allows that to happen.

Linux implements dynamic linking in user mode through a special linker library. Every dynamically linked program contains a small, statically linked function that is called when the program starts. This static function just maps the link library into memory and runs the code that the function contains. The link library determines the dynamic libraries required by the program and the names of the variables and functions needed from those libraries by reading the information contained in sections of the ELF binary. It then maps the libraries into the middle of virtual memory and resolves the references to the symbols contained in those libraries. It does not matter exactly where in memory these shared libraries are mapped: they are compiled into **position-independent code** (**PIC**), which can run at any address in memory.

## 20.7 File Systems

Linux retains UNIX's standard file-system model. In UNIX, a file does not have to be an object stored on disk or fetched over a network from a remote file server. Rather, UNIX files can be anything capable of handling the input or output of a stream of data. Device drivers can appear as files, and interprocess-communication channels or network connections also look like files to the user.

The Linux kernel handles all these types of files by hiding the implementation details of any single file type behind a layer of software, the virtual file system (VFS). Here, we first cover the virtual file system and then discuss the standard Linux file system—ext3.

### 20.7.1 The Virtual File System

The Linux VFS is designed around object-oriented principles. It has two components: a set of definitions that specify what file-system objects are allowed to look like and a layer of software to manipulate the objects. The VFS defines four main object types:

- An **inode object** represents an individual file.
- A **file object** represents an open file.
- A **superblock object** represents an entire file system.
- A **dentry object** represents an individual directory entry.

For each of these four object types, the VFS defines a set of operations. Every object of one of these types contains a pointer to a function table. The function table lists the addresses of the actual functions that implement the defined operations for that object. For example, an abbreviated API for some of the file object's operations includes:

- `int open(. . .)` — Open a file.
- `ssize_t read(. . .)` — Read from a file.
- `ssize_t write(. . .)` — Write to a file.
- `int mmap(. . .)` — Memory-map a file.

The complete definition of the file object is specified in the `struct file_operations`, which is located in the file `/usr/include/linux/fs.h`. An implementation of the file object (for a specific file type) is required to implement each function specified in the definition of the file object.

The VFS software layer can perform an operation on one of the file-system objects by calling the appropriate function from the object's function table, without having to know in advance exactly what kind of object it is dealing with. The VFS does not know, or care, whether an inode represents a networked file, a disk file, a network socket, or a directory file. The appropriate function for that file's `read()` operation will always be at the same place in its function table, and the VFS software layer will call that function without caring how the data are actually read.

The inode and file objects are the mechanisms used to access files. An inode object is a data structure containing pointers to the disk blocks that contain the actual file contents, and a file object represents a point of access to the data in an open file. A thread cannot access an inode's contents without first obtaining a file object pointing to the inode. The file object keeps track of where in the file the process is currently reading or writing, to keep track of sequential file I/O. It also remembers the permissions (for example, read or write) requested when the file was opened and tracks the thread's activity if necessary to perform adaptive read-ahead, fetching file data into memory before the thread requests the data, to improve performance.

File objects typically belong to a single process, but inode objects do not. There is one file object for every instance of an open file, but always only a single inode object. Even when a file is no longer in use by any process, its inode object may still be cached by the VFS to improve performance if the file is used again in the near future. All cached file data are linked onto a list in the file's inode object. The inode also maintains standard information about each file, such as the owner, size, and time most recently modified.

Directory files are dealt with slightly differently from other files. The UNIX programming interface defines a number of operations on directories, such as creating, deleting, and renaming a file in a directory. The system calls for these directory operations do not require that the user open the files concerned, unlike the case for reading or writing data. The VFS therefore defines these directory operations in the inode object, rather than in the file object.

The superblock object represents a connected set of files that form a self-contained file system. The operating-system kernel maintains a single superblock object for each disk device mounted as a file system and for each networked file system currently connected. The main responsibility of the superblock object is to provide access to inodes. The VFS identifies every inode by a unique file-system/inode number pair, and it finds the inode corresponding to a particular inode number by asking the superblock object to return the inode with that number.

Finally, a dentry object represents a directory entry, which may include the name of a directory in the path name of a file (such as `/usr`) or the actual file (such as `stdio.h`). For example, the file `/usr/include/stdio.h` contains the directory entries (1) `/`, (2) `usr`, (3) `include`, and (4) `stdio.h`. Each of these values is represented by a separate dentry object.

As an example of how dentry objects are used, consider the situation in which a thread wishes to open the file with the pathname `/usr/include/stdio.h` using an editor. Because Linux treats directory names as files, translating this path requires first obtaining the inode for the root—`/`. The operating system must then read through this file to obtain the inode for the file `include`. It must continue this thread until it obtains the inode for the file `stdio.h`. Because path-name translation can be a time-consuming task, Linux maintains a cache of dentry objects, which is consulted during path-name translation. Obtaining the inode from the dentry cache is considerably faster than having to read the on-disk file.

### 20.7.2 The Linux ext3 File System

The standard on-disk file system used by Linux is called **ext3**, for historical reasons. Linux was originally programmed with a Minix-compatible file system, to ease exchanging data with the Minix development system, but that file system was severely restricted by 14-character file-name limits and a maximum file-system size of 64-MB. The Minix file system was superseded by a new file system, which was christened the **extended file system** (**extfs**). A later redesign to improve performance and scalability and to add a few missing features led to the **second extended file system** (**ext2**). Further development added journaling capabilities, and the system was renamed the **third extended file system** (**ext3**). Linux kernel developers then augmented ext3 with modern file-system features such as extents. This new file system is called the **fourth extended file system** (**ext4**). The rest of this section discusses ext3, however, since it remains the most-deployed Linux file system. Most of the discussion applies equally to ext4.

Linux's ext3 has much in common with the BSD Fast File System (FFS) (Section C.7.7). It uses a similar mechanism for locating the data blocks belonging to a specific file, storing data-block pointers in indirect blocks throughout the file system with up to three levels of indirection. As in FFS, directory files are stored on disk just like normal files, although their contents are interpreted differently. Each block in a directory file consists of a linked list of entries. In turn, each entry contains the length of the entry, the name of a file, and the inode number of the inode to which that entry refers.

The main differences between ext3 and FFS lie in their disk-allocation policies. In FFS, the disk is allocated to files in blocks of 8 KB. These blocks are subdivided into fragments of 1 KB for storage of small files or partially filled blocks at the ends of files. In contrast, ext3 does not use fragments at all but performs all its allocations in smaller units. The default block size on ext3 varies as a function of the total size of the file system. Supported block sizes are 1, 2, 4, and 8 KB.

To maintain high performance, the operating system must try to perform I/O operations in large chunks whenever possible by clustering physically adjacent I/O requests. Clustering reduces the per-request overhead incurred by device drivers, disks, and disk-controller hardware. A block-sized I/O request size is too small to maintain good performance, so ext3 uses allocation policies designed to place logically adjacent blocks of a file into physically adjacent blocks on disk, so that it can submit an I/O request for several disk blocks as a single operation.

The ext3 allocation policy works as follows: As in FFS, an ext3 file system is partitioned into multiple segments. In ext3, these are called **block groups**. FFS uses the similar concept of **cylinder groups**, where each group corresponds to a single cylinder of a physical disk. (Note that modern disk-drive technology packs sectors onto the disk at different densities, and thus with different cylinder sizes, depending on how far the disk head is from the center of the disk. Therefore, fixed-sized cylinder groups do not necessarily correspond to the disk's geometry.)

When allocating a file, ext3 must first select the block group for that file. For data blocks, it attempts to allocate the file to the block group to which the file's inode has been allocated. For inode allocations, it selects the block group in which the file's parent directory resides for nondirectory files. Directory files are not kept together but rather are dispersed throughout the available block groups. These policies are designed not only to keep related information within the same block group but also to spread out the disk load among the disk's block groups to reduce the fragmentation of any one area of the disk.

Within a block group, ext3 tries to keep allocations physically contiguous if possible, reducing fragmentation if it can. It maintains a bitmap of all free blocks in a block group. When allocating the first blocks for a new file, it starts searching for a free block from the beginning of the block group. When extending a file, it continues the search from the block most recently allocated to the file. The search is performed in two stages. First, ext3 searches for an entire free byte in the bitmap; if it fails to find one, it looks for any free bit. The search for free bytes aims to allocate disk space in chunks of at least eight blocks where possible.

Once a free block has been identified, the search is extended backward until an allocated block is encountered. When a free byte is found in the bitmap, this backward extension prevents ext3 from leaving a hole between the most recently allocated block in the previous nonzero byte and the zero byte found. Once the next block to be allocated has been found by either bit or byte search, ext3 extends the allocation forward for up to eight blocks and preallocates these extra blocks to the file. This preallocation helps to reduce fragmentation during interleaved writes to separate files and also reduces the CPU cost of disk allocation by allocating multiple blocks simultaneously. The preallocated blocks are returned to the free-space bitmap when the file is closed.

Figure 20.7 illustrates the allocation policies. Each row represents a sequence of set and unset bits in an allocation bitmap, indicating used and free blocks on disk. In the first case, if we can find any free blocks sufficiently near the start of the search, then we allocate them no matter how fragmented they may be. The fragmentation is partially compensated for by the fact that the blocks are close together and can probably all be read without any disk seeks. Furthermore, allocating them all to one file is better in the long run than allocating isolated blocks to separate files once large free areas become scarce on disk. In the second case, we have not immediately found a free block close by, so we search forward for an entire free byte in the bitmap. If we allocated that byte as a whole, we would end up creating a fragmented area of free space between it and the allocation preceding it. Thus, before allocating, we back up to make this allocation flush with the allocation preceding it, and then we allocate forward to satisfy the default allocation of eight blocks.

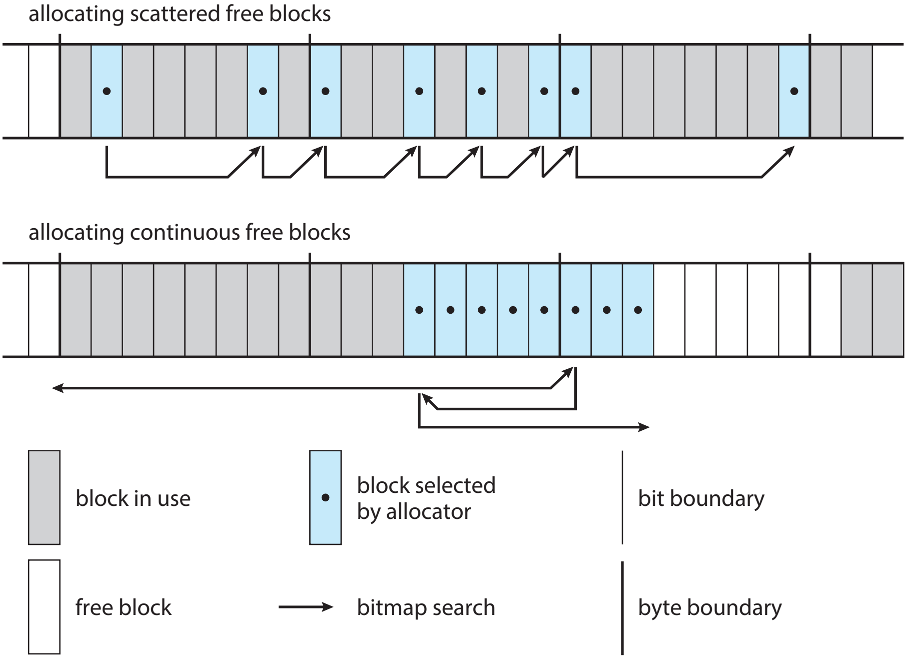

allocating scattered free blocks
allocating continuous free blocks
block in use
block selected by allocator
bit boundary
free block
bitmap search
byte boundary

**Figure 20.7** ext3 block-allocation policies.

### 20.7.3 Journaling

The ext3 file system supports a popular feature called **journaling**, whereby modifications to the file system are written sequentially to a journal. A set of operations that performs a specific task is a **transaction**. Once a transaction is written to the journal, it is considered to be committed. Meanwhile, the journal entries relating to the transaction are replayed across the actual file-system structures. As the changes are made, a pointer is updated to indicate which actions have completed and which are still incomplete. When an entire committed transaction is completed, it is removed from the journal. The journal, which is actually a circular buffer, may be in a separate section of the file system, or it may even be on a separate disk spindle. It is more efficient, but more complex, to have it under separate read–write heads, thereby decreasing head contention and seek times.

If the system crashes, some transactions may remain in the journal. Those transactions were never completed to the file system even though they were committed by the operating system, so they must be completed once the system recovers. The transactions can be executed from the pointer until the work is complete, and the file-system structures remain consistent. The only problem occurs when a transaction has been aborted—that is, it was not committed before the system crashed. Any changes from those transactions that were applied to the file system must be undone, again preserving the consistency of the file system. This recovery is all that is needed after a crash, eliminating all problems with consistency checking.

Journaling file systems may perform some operations faster than nonjournaling systems, as updates proceed much faster when they are applied to the in-memory journal rather than directly to the on-disk data structures. The reason for this improvement is found in the performance advantage of sequential I/O over random I/O. Costly synchronous random writes to the file system are turned into much less costly synchronous sequential writes to the file system's journal. Those changes, in turn, are replayed asynchronously via random writes to the appropriate structures. The overall result is a significant gain in performance of file-system metadata-oriented operations, such as file creation and deletion. Due to this performance improvement, ext3 can be configured to journal only metadata and not file data.

### 20.7.4 The Linux Proc File System

The flexibility of the Linux VFS enables us to implement a file system that does not store data persistently at all but rather provides an interface to some other functionality. The Linux `/proc` file system is an example of a file system whose contents are not actually stored anywhere but are computed on demand according to user file I/O requests.

A `/proc` file system is not unique to Linux. UNIX v8 introduced a `/proc` file system and its use has been adopted and expanded into many other operating systems. It is an efficient interface to the kernel's process name space and helps with debugging. Each subdirectory of the file system corresponded not to a directory on any disk but rather to an active process on the current system. A listing of the file system reveals one directory per process, with the directory name being the ASCII decimal representation of the process's unique process identifier (PID).

Linux implements such a `/proc` file system but extends it greatly by adding a number of extra directories and text files under the file system's root directory. These new entries correspond to various statistics about the kernel and the associated loaded drivers. The `/proc` file system provides a way for programs to access this information as plain text files; the standard UNIX user environment provides powerful tools to process such files. For example, in the past, the traditional UNIX `ps` command for listing the states of all running processes has been implemented as a privileged process that reads the process state directly from the kernel's virtual memory. Under Linux, this command is implemented as an entirely unprivileged program that simply parses and formats the information from `/proc`.

The `/proc` file system must implement two things: a directory structure and the file contents within. Because a UNIX file system is defined as a set of file and directory inodes identified by their inode numbers, the `/proc` file system must define a unique and persistent inode number for each directory and the associated files. Once such a mapping exists, the file system can use this inode number to identify just what operation is required when a user tries to read from a particular file inode or to perform a lookup in a particular directory inode. When data are read from one of these files, the `/proc` file system will collect the appropriate information, format it into textual form, and place it into the requesting process's read buffer.

The mapping from inode number to information type splits the inode number into two fields. In Linux, a PID is 16 bits in size, but an inode number is 32 bits. The top 16 bits of the inode number are interpreted as a PID, and the remaining bits define what type of information is being requested about that process.

A PID of zero is not valid, so a zero PID field in the inode number is taken to mean that this inode contains global—rather than process-specific—information. Separate global files exist in `/proc` to report information such as the kernel version, free memory, performance statistics, and drivers currently running.

Not all the inode numbers in this range are reserved. The kernel can allocate new `/proc` inode mappings dynamically, maintaining a bitmap of allocated inode numbers. It also maintains a tree data structure of registered global `/proc` file-system entries. Each entry contains the file's inode number, file name, and access permissions, along with the special functions used to generate the file's contents. Drivers can register and deregister entries in this tree at any time, and a special section of the tree—appearing under the `/proc/sys` directory—is reserved for kernel variables. Files under this tree are managed by a set of common handlers that allow both reading and writing of these variables, so a system administrator can tune the value of kernel parameters simply by writing out the new desired values in ASCII decimal to the appropriate file.

To allow efficient access to these variables from within applications, the `/proc/sys` subtree is made available through a special system call, `sysctl()`, that reads and writes the same variables in binary, rather than in text, without the overhead of the file system. `sysctl()` is not an extra facility; it simply reads the `/proc` dynamic entry tree to identify the variables to which the application is referring.

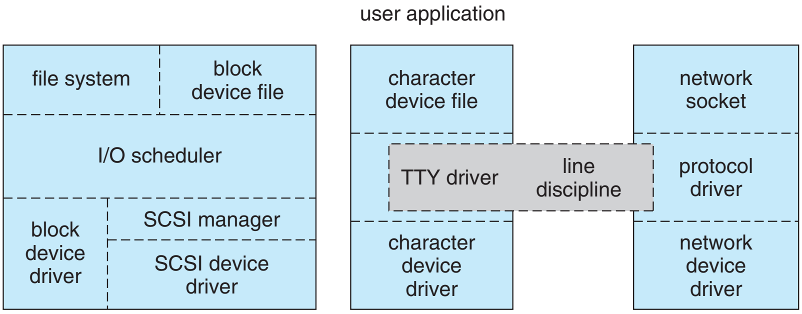


**Figure 20.8** Device-driver block structure.

## 20.8 Input and Output

To the user, the I/O system in Linux looks much like that in any UNIX system. That is, to the extent possible, all device drivers appear as normal files. Users can open an access channel to a device in the same way they open any other file—devices can appear as objects within the file system. The system administrator can create special files within a file system that contain references to a specific device driver, and a user opening such a file will be able to read from and write to the device referenced. By using the normal file-protection system, which determines who can access which file, the administrator can set access permissions for each device.

Linux splits all devices into three classes: block devices, character devices, and network devices. Figure 20.8 illustrates the overall structure of the device-driver system.

**Block devices** include all devices that allow random access to completely independent, fixed-sized blocks of data, including hard disks and floppy disks, CD-ROMs and Blu-ray discs, and flash memory. Block devices are typically used to store file systems, but direct access to a block device is also allowed so that programs can create and repair the file system that the device contains. Applications can also access these block devices directly if they wish. For example, a database application may prefer to perform its own fine-tuned layout of data onto a disk rather than using the general-purpose file system.

**Character devices** include most other devices, such as mice and keyboards. The fundamental difference between block and character devices is random access—block devices are accessed randomly, while character devices are accessed serially. For example, seeking to a certain position in a file might be supported for a DVD but makes no sense for a pointing device such as a mouse.

**Network devices** are dealt with differently from block and character devices. Users cannot directly transfer data to network devices. Instead, they must communicate indirectly by opening a connection to the kernel's networking subsystem. We discuss the interface to network devices separately in Section 20.10.

### 20.8.1 Block Devices

Block devices provide the main interface to all disk devices in a system. Performance is particularly important for disks, and the block-device system must provide functionality to ensure that disk access is as fast as possible. This functionality is achieved through the scheduling of I/O operations.

In the context of block devices, a block represents the unit with which the kernel performs I/O. When a block is read into memory, it is stored in a buffer. The **request manager** is the layer of software that manages the reading and writing of buffer contents to and from a block-device driver.

A separate list of requests is kept for each block-device driver. Traditionally, these requests have been scheduled according to a unidirectional-elevator (C-SCAN) algorithm that exploits the order in which requests are inserted in and removed from the lists. The request lists are maintained in sorted order of increasing starting-sector number. When a request is accepted for processing by a block-device driver, it is not removed from the list. It is removed only after the I/O is complete, at which point the driver continues with the next request in the list, even if new requests have been inserted in the list before the active request. As new I/O requests are made, the request manager attempts to merge requests in the lists.

Linux kernel version 2.6 introduced a new I/O scheduling algorithm. Although a simple elevator algorithm remains available, the default I/O scheduler is now the **Completely Fair Queueing** (**CFQ**) scheduler. The CFQ I/O scheduler is fundamentally different from elevator-based algorithms. Instead of sorting requests into a list, CFQ maintains a set of lists—by default, one for each process. Requests originating from a process go in that process's list. For example, if two processes are issuing I/O requests, CFQ will maintain two separate lists of requests, one for each process. The lists are maintained according to the C-SCAN algorithm.

CFQ services the lists differently as well. Where a traditional C-SCAN algorithm is indifferent to a specific process, CFQ services each process's list round-robin. It pulls a configurable number of requests (by default, four) from each list before moving on to the next. This method results in fairness at the process level—each process receives an equal fraction of the disk's bandwidth. The result is beneficial with interactive workloads where I/O latency is important. In practice, however, CFQ performs well with most workloads.

### 20.8.2 Character Devices

A character-device driver can be almost any device driver that does not offer random access to fixed blocks of data. Any character-device drivers registered to the Linux kernel must also register a set of functions that implement the file I/O operations that the driver can handle. The kernel performs almost no preprocessing of a file read or write request to a character device. It simply passes the request to the device in question and lets the device deal with the request.

The main exception to this rule is the special subset of character-device drivers that implement terminal devices. The kernel maintains a standard interface to these drivers by means of a set of `tty_struct` structures. Each of these structures provides buffering and flow control on the data stream from the terminal device and feeds those data to a line discipline.

A **line discipline** is an interpreter for the information from the terminal device. The most common line discipline is the `tty` discipline, which glues the terminal's data stream onto the standard input and output streams of a user's running processes, allowing those processes to communicate directly with the user's terminal. This job is complicated by the fact that several such processes may be running simultaneously, and the `tty` line discipline is responsible for attaching and detaching the terminal's input and output from the various processes connected to it as those processes are suspended or awakened by the user.

Other line disciplines also are implemented that have nothing to do with I/O to a user process. The PPP and SLIP networking protocols are ways of encoding a networking connection over a terminal device such as a serial line. These protocols are implemented under Linux as drivers that at one end appear to the terminal system as line disciplines and at the other end appear to the networking system as network-device drivers. After one of these line disciplines has been enabled on a terminal device, any data appearing on that terminal will be routed directly to the appropriate network-device driver.

## 20.9 Interprocess Communication

Linux provides a rich environment for processes to communicate with each other. Communication may be just a matter of letting another process know that some event has occurred, or it may involve transferring data from one process to another.

### 20.9.1 Synchronization and Signals

The standard Linux mechanism for informing a process that an event has occurred is the **signal**. Signals can be sent from any process to any other process, with restrictions on signals sent to processes owned by another user. However, a limited number of signals is available, and they cannot carry information. Only the fact that a signal has occurred is available to a process. Signals are not generated only by processes. The kernel also generates signals internally. For example, it can send a signal to a server process when data arrive on a network channel, to a parent process when a child terminates, or to a waiting process when a timer expires.

Internally, the Linux kernel does not use signals to communicate with processes running in kernel mode. If a kernel-mode process is expecting an event to occur, it will not use signals to receive notification of that event. Rather, communication about incoming asynchronous events within the kernel takes place through the use of scheduling states and `wait_queue` structures. These mechanisms allow kernel-mode processes to inform one another about relevant events, and they also allow events to be generated by device drivers or by the networking system. Whenever a process wants to wait for some event to complete, it places itself on a wait queue associated with that event and tells the scheduler that it is no longer eligible for execution. Once the event has completed, every process on the wait queue will be awakened. This procedure allows multiple processes to wait for a single event. For example, if several processes are trying to read a file from a disk, then they will all be awakened once the data have been read into memory successfully.

Although signals have always been the main mechanism for communicating asynchronous events among processes, Linux also implements the semaphore mechanism of System V UNIX. A process can wait on a semaphore as easily as it can wait for a signal, but semaphores have two advantages: large numbers of semaphores can be shared among multiple independent processes, and operations on multiple semaphores can be performed atomically. Internally, the standard Linux wait queue mechanism synchronizes processes that are communicating with semaphores.

### 20.9.2 Passing of Data among Processes

Linux offers several mechanisms for passing data among processes. The standard UNIX **pipe** mechanism allows a child process to inherit a communication channel from its parent; data written to one end of the pipe can be read at the other. Under Linux, pipes appear as just another type of inode to virtual file system software, and each pipe has a pair of wait queues to synchronize the reader and writer. UNIX also defines a set of networking facilities that can send streams of data to both local and remote processes. Networking is covered in Section 20.10.

Another process communications method, shared memory, offers an extremely fast way to communicate large or small amounts of data. Any data written by one process to a shared memory region can be read immediately by any other process that has mapped that region into its address space. The main disadvantage of shared memory is that, on its own, it offers no synchronization. A process can neither ask the operating system whether a piece of shared memory has been written to nor suspend execution until such a write occurs. Shared memory becomes particularly powerful when used in conjunction with another interprocess-communication mechanism that provides the missing synchronization.

A shared-memory region in Linux is a persistent object that can be created or deleted by processes. Such an object is treated as though it were a small, independent address space. The Linux paging algorithms can elect to page shared-memory pages out to disk, just as they can page out a process's data pages. The shared-memory object acts as a backing store for shared-memory regions, just as a file can act as a backing store for a memory-mapped memory region. When a file is mapped into a virtual address space region, then any page faults that occur cause the appropriate page of the file to be mapped into virtual memory. Similarly, shared-memory mappings direct page faults to map in pages from a persistent shared-memory object. Also just as for files, shared-memory objects remember their contents even if no processes are currently mapping them into virtual memory.

## 20.10 Network Structure

Networking is a key area of functionality for Linux. Not only does Linux support the standard Internet protocols used for most UNIX-to-UNIX communications, but it also implements a number of protocols native to other, non-UNIX operating systems. In particular, since Linux was originally implemented primarily on PCs, rather than on large workstations or on server-class systems, it supports many of the protocols typically used on PC networks, such as AppleTalk and IPX.

Internally, networking in the Linux kernel is implemented by three layers of software:

1. The socket interface
2. Protocol drivers
3. Network-device drivers

User applications perform all networking requests through the socket interface. This interface is designed to look like the 4.3 BSD socket layer, so that any programs designed to make use of Berkeley sockets will run on Linux without any source-code changes. This interface is described in Section C.9.1. The BSD socket interface is sufficiently general to represent network addresses for a wide range of networking protocols. This single interface is used in Linux to access not just those protocols implemented on standard BSD systems but all the protocols supported by the system.

The next layer of software is the protocol stack, which is similar in organization to BSD's own framework. Whenever any networking data arrive at this layer, either from an application's socket or from a network-device driver, the data are expected to have been tagged with an identifier specifying which network protocol they contain. Protocols can communicate with one another if they desire; for example, within the Internet protocol set, separate protocols manage routing, error reporting, and reliable retransmission of lost data.

The protocol layer may rewrite packets, create new packets, split or reassemble packets into fragments, or simply discard incoming data. Ultimately, once the protocol layer has finished processing a set of packets, it passes them on, either upward to the socket interface if the data are destined for a local connection or downward to a device driver if the data need to be transmitted remotely. The protocol layer decides to which socket or device it will send the packet.

All communication between the layers of the networking stack is performed by passing single `skbuff` (socket buffer) structures. Each of these structures contains a set of pointers into a single continuous area of memory, representing a buffer inside which network packets can be constructed. The valid data in a `skbuff` do not need to start at the beginning of the `skbuff`'s buffer, and they do not need to run to the end. The networking code can add data to or trim data from either end of the packet, as long as the result still fits into the `skbuff`. This capacity is especially important on modern microprocessors, where improvements in CPU speed have far outstripped the performance of main memory. The `skbuff` architecture allows flexibility in manipulating packet headers and checksums while avoiding any unnecessary data copying.

The most important set of protocols in the Linux networking system is the TCP/IP protocol suite. This suite comprises a number of separate protocols. The IP protocol implements routing between different hosts anywhere on the network. On top of the routing protocol are the UDP, TCP, and ICMP protocols. The UDP protocol carries arbitrary individual datagrams between hosts. The TCP protocol implements reliable connections between hosts with guaranteed in-order delivery of packets and automatic retransmission of lost data. The ICMP protocol carries various error and status messages between hosts.

Each packet (`skbuff`) arriving at the networking stack's protocol software is expected to be already tagged with an internal identifier indicating the protocol to which the packet is relevant. Different networking-device drivers encode the protocol type in different ways; thus, the protocol for incoming data must be identified in the device driver. The device driver uses a hash table of known networking-protocol identifiers to look up the appropriate protocol and passes the packet to that protocol. New protocols can be added to the hash table as kernel-loadable modules.

Incoming IP packets are delivered to the IP driver. The job of this layer is to perform routing. After deciding where the packet is to be sent, the IP driver forwards the packet to the appropriate internal protocol driver to be delivered locally or injects it back into a selected network-device-driver queue to be forwarded to another host. It performs the routing decision using two tables: the persistent forwarding information base (FIB) and a cache of recent routing decisions. The FIB holds routing-configuration information and can specify routes based either on a specific destination address or on a wildcard representing multiple destinations. The FIB is organized as a set of hash tables indexed by destination address; the tables representing the most specific routes are always searched first. Successful lookups from this table are added to the route-caching table, which caches routes only by specific destination. No wildcards are stored in the cache, so lookups can be made quickly. An entry in the route cache expires after a fixed period with no hits.

At various stages, the IP software passes packets to a separate section of code for **firewall management**—selective filtering of packets according to arbitrary criteria, usually for security purposes. The firewall manager maintains a number of separate **firewall chains** and allows a `skbuff` to be matched against any chain. Chains are reserved for separate purposes: one is used for forwarded packets, one for packets being input to this host, and one for data generated at this host. Each chain is held as an ordered list of rules, where a rule specifies one of a number of possible firewall-decision functions plus some arbitrary data for matching purposes.

Two other functions performed by the IP driver are disassembly and reassembly of large packets. If an outgoing packet is too large to be queued to a device, it is simply split up into smaller **fragments**, which are all queued to the driver. At the receiving host, these fragments must be reassembled. The IP driver maintains an `ipfrag` object for each fragment awaiting reassembly and an `ipq` for each datagram being assembled. Incoming fragments are matched against each known `ipq`. If a match is found, the fragment is added to it; otherwise, a new `ipq` is created. Once the final fragment has arrived for a `ipq`, a completely new `skbuff` is constructed to hold the new packet, and this packet is passed back into the IP driver.

Packets identified by the IP as destined for this host are passed on to one of the other protocol drivers. The UDP and TCP protocols share a means of associating packets with source and destination sockets: each connected pair of sockets is uniquely identified by its source and destination addresses and by the source and destination port numbers. The socket lists are linked to hash tables keyed on these four address and port values for socket lookup on incoming packets. The TCP protocol has to deal with unreliable connections, so it maintains ordered lists of unacknowledged outgoing packets to retransmit after a timeout and of incoming out-of-order packets to be presented to the socket when the missing data have arrived.

## 20.11 Security

Linux's security model is closely related to typical UNIX security mechanisms. The security concerns can be classified in two groups:

1. **Authentication**. Making sure that nobody can access the system without first proving that she has entry rights
2. **Access control**. Providing a mechanism for checking whether a user has the right to access a certain object and preventing access to objects as required

### 20.11.1 Authentication

Authentication in UNIX has typically been performed through the use of a publicly readable password file. A user's password is combined with a random "salt" value, and the result is encoded with a one-way transformation function and stored in the password file. The use of the one-way function means that the original password cannot be deduced from the password file except by trial and error. When a user presents a password to the system, the password is recombined with the salt value stored in the password file and passed through the same one-way transformation. If the result matches the contents of the password file, then the password is accepted.

Historically, UNIX implementations of this mechanism have had several drawbacks. Passwords were often limited to eight characters, and the number of possible salt values was so low that an attacker could easily combine a dictionary of commonly used passwords with every possible salt value and have a good chance of matching one or more passwords in the password file, gaining unauthorized access to any accounts compromised as a result. Extensions to the password mechanism have been introduced that keep the encrypted password secret in a file that is not publicly readable, that allow longer passwords, or that use more secure methods of encoding the password. Other authentication mechanisms have been introduced that limit the periods during which a user is permitted to connect to the system. Also, mechanisms exist to distribute authentication information to all the related systems in a network.

A new security mechanism has been developed by UNIX vendors to address authentication problems. The **pluggable authentication modules** (**PAM**) system is based on a shared library that can be used by any system component that needs to authenticate users. An implementation of this system is available under Linux. PAM allows authentication modules to be loaded on demand as specified in a system-wide configuration file. If a new authentication mechanism is added at a later date, it can be added to the configuration file, and all system components will immediately be able to take advantage of it. PAM modules can specify authentication methods, account restrictions, session-setup functions, and password-changing functions (so that, when users change their passwords, all the necessary authentication mechanisms can be updated at once).

### 20.11.2 Access Control

Access control under UNIX systems, including Linux, is performed through the use of unique numeric identifiers. A user identifier (UID) identifies a single user or a single set of access rights. A group identifier (GID) is an extra identifier that can be used to identify rights belonging to more than one user.

Access control is applied to various objects in the system. Every file available in the system is protected by the standard access-control mechanism. In addition, other shared objects, such as shared-memory sections and semaphores, employ the same access system.

Every object in a UNIX system under user and group access control has a single UID and a single GID associated with it. User processes also have a single UID, but they may have more than one GID. If a process's UID matches the UID of an object, then the process has **user rights** or **owner rights** to that object. If the UIDs do not match but any GID of the process matches the object's GID, then **group rights** are conferred; otherwise, the process has **world rights** to the object.

Linux performs access control by assigning objects a **protection mask** that specifies which access modes—read, write, or execute—are to be granted to processes with owner, group, or world access. Thus, the owner of an object might have full read, write, and execute access to a file; other users in a certain group might be given read access but denied write access; and everybody else might be given no access at all.

The only exception is the privileged ***root*** UID. A process with this special UID is granted automatic access to any object in the system, bypassing normal access checks. Such processes are also granted permission to perform privileged operations, such as reading any physical memory or opening reserved network sockets. This mechanism allows the kernel to prevent normal users from accessing these resources: most of the kernel's key internal resources are implicitly owned by the root UID.

Linux implements the standard UNIX `setuid` mechanism described in Section C.3.2. This mechanism allows a program to run with privileges different from those of the user running the program. For example, the `lpr` program (which submits a job to a print queue) has access to the system's print queues even if the user running that program does not. The UNIX implementation of `setuid` distinguishes between a process's `real` and `effective` UID. The real UID is that of the user running the program; the effective UID is that of the file's owner.

Under Linux, this mechanism is augmented in two ways. First, Linux implements the POSIX specification's `saved user-id` mechanism, which allows a process to drop and reacquire its effective UID repeatedly. For security reasons, a program may want to perform most of its operations in a safe mode, waiving the privileges granted by its `setuid` status; but it may wish to perform selected operations with all its privileges. Standard UNIX implementations achieve this capacity only by swapping the real and effective UIDs. When this is done, the previous effective UID is remembered, but the program's real UID does not always correspond to the UID of the user running the program. Saved UIDs allow a process to set its effective UID to its real UID and then return to the previous value of its effective UID without having to modify the real UID at any time.

The second enhancement provided by Linux is the addition of a process characteristic that grants just a subset of the rights of the effective UID. The **fsuid** and **fsgid** process properties are used when access rights are granted to files. The appropriate property is set every time the effective UID or GID is set. However, the fsuid and fsgid can be set independently of the effective ids, allowing a process to access files on behalf of another user without taking on the identity of that other user in any other way. Specifically, server processes can use this mechanism to serve files to a certain user without becoming vulnerable to being killed or suspended by that user.

Finally, Linux provides a mechanism for flexible passing of rights from one program to another—a mechanism that has become common in modern versions of UNIX. When a local network socket has been set up between any two processes on the system, either of those processes may send to the other process a file descriptor for one of its open files; the other process receives a duplicate file descriptor for the same file. This mechanism allows a client to pass access to a single file selectively to some server process without granting that process any other privileges. For example, it is no longer necessary for a print server to be able to read all the files of a user who submits a new print job. The print client can simply pass the server file descriptors for any files to be printed, denying the server access to any of the user's other files.

## 20.12 Summary

- Linux is a modern, free operating system based on UNIX standards. It has been designed to run efficiently and reliably on common PC hardware; it also runs on a variety of other platforms, such as mobile phones. It provides a programming interface and user interface compatible with standard UNIX systems and can run a large number of UNIX applications, including an increasing number of commercially supported applications.
- Linux has not evolved in a vacuum. A complete Linux system includes many components that were developed independently of Linux. The core Linux operating-system kernel is entirely original, but it allows much existing free UNIX software to run, resulting in an entire UNIX-compatible operating system free from proprietary code.
- The Linux kernel is implemented as a traditional monolithic kernel for performance reasons, but it is modular enough in design to allow most drivers to be dynamically loaded and unloaded at run time.
- Linux is a multiuser system, providing protection between processes and running multiple processes according to a time-sharing scheduler. Newly created processes can share selective parts of their execution environment with their parent processes, allowing multithreaded programming.
- Interprocess communication is supported by both System V mechanisms —message queues, semaphores, and shared memory—and BSD's socket interface. Multiple networking protocols can be accessed simultaneously through the socket interface.

- The memory-management system uses page sharing and copy-on-write to minimize the duplication of data shared by different processes. Pages are loaded on demand when they are first referenced and are paged back out to backing store according to an LFU algorithm if physical memory needs to be reclaimed.
- To the user, the file system appears as a hierarchical directory tree that obeys UNIX semantics. Internally, Linux uses an abstraction layer to manage multiple file systems. Device-oriented, networked, and virtual file systems are supported. Device-oriented file systems access disk storage through a page cache that is unified with the virtual memory system.

## Practice Exercises

**20.1** Dynamically loadable kernel modules give flexibility when drivers are added to a system, but do they have disadvantages too? Under what circumstances would a kernel be compiled into a single binary file, and when would it be better to keep it split into modules? Explain your answer.

**20.2** Multithreading is a commonly used programming technique. Describe three different ways to implement threads, and compare these three methods with the Linux `clone()` mechanism. When might using each alternative mechanism be better or worse than using clones?

**20.3** The Linux kernel does not allow paging out of kernel memory. What effect does this restriction have on the kernel's design? What are two advantages and two disadvantages of this design decision?

**20.4** Discuss three advantages of dynamic (shared) linkage of libraries compared with static linkage. Describe two cases in which static linkage is preferable.

**20.5** Compare the use of networking sockets with the use of shared memory as a mechanism for communicating data between processes on a single computer. What are the advantages of each method? When might each be preferred?

**20.6** At one time, UNIX systems used disk-layout optimizations based on the rotation position of disk data, but modern implementations, including Linux, simply optimize for sequential data access. Why do they do so? Of what hardware characteristics does sequential access take advantage? Why is rotational optimization no longer so useful?

## Further Reading

The Linux system is a product of the Internet; as a result, much of the available documentation on Linux is available in some form on the Internet. The following key sites reference most of the useful information available:

- The *Linux Cross-Reference Page (LXR)* (http://lxr.linux.no) maintains current listings of the Linux kernel, browsable via the web and fully cross-referenced.
- The *Kernel Hackers' Guide* provides a helpful overview of the Linux kernel components and internals and is located at http://tldp.org/LDP/tlk/tlk.html.
- The *Linux Weekly News (LWN)* (http://lwn.net) provides weekly Linux-related news, including a very well researched subsection on Linux kernel news.

Many mailing lists devoted to Linux are also available. The most important are maintained by a mailing-list manager that can be reached at the e-mail address majordomo@vger.rutgers.edu. Send e-mail to this address with the single line "help" in the mail's body for information on how to access the list server and to subscribe to any lists.

Finally, the Linux system itself can be obtained over the Internet. Complete Linux distributions are available from the home sites of the companies concerned, and the Linux community also maintains archives of current system components at several places on the Internet. The most important is ftp://ftp.kernel.org/pub/linux.

In addition to investigating Internet resources, you can read about the internals of the Linux kernel in [Mauerer (2008)] and [Love (2010)].

The `/proc` file system was introduced in http://lucasvr.gobolinux.org/etc/Killian84-Procfs-USENIX.pdf, and expanded in http://https://www.usenix.org/sites/default/files/usenix_winter91_faulkner.pdf.

## Bibliography

**[Love (2010)]** R. Love, *Linux Kernel Development*, Third Edition, Developer's Library (2010).

**[Mauerer (2008)]** W. Mauerer, *Professional Linux Kernel Architecture*, John Wiley and Sons (2008).

## Chapter 20 Exercises

**20.7** What are the advantages and disadvantages of writing an operating system in a high-level language, such as C?

**20.8** In what circumstances is the system-call sequence `fork()` `exec()` most appropriate? When is `vfork()` preferable?

**20.9** What socket type should be used to implement an intercomputer file-transfer program? What type should be used for a program that periodically tests to see whether another computer is up on the network? Explain your answer.

**20.10** Linux runs on a variety of hardware platforms. What steps must Linux developers take to ensure that the system is portable to different processors and memory-management architectures and to minimize the amount of architecture-specific kernel code?

**20.11** What are the advantages and disadvantages of making only some of the symbols defined inside a kernel accessible to a loadable kernel module?

**20.12** What are the primary goals of the conflict-resolution mechanism used by the Linux kernel for loading kernel modules?

**20.13** Discuss how the clone() operation supported by Linux is used to support both processes and threads.

**20.14** Would you classify Linux threads as user-level threads or as kernel-level threads? Support your answer with the appropriate arguments.

**20.15** What extra costs are incurred in the creation and scheduling of a process, compared with the cost of a cloned thread?

**20.16** How does Linux's Completely Fair Scheduler (CFS) provide improved fairness over a traditional UNIX process scheduler? When is the fairness guaranteed?

**20.17** What are the two configurable variables of the Completely Fair Scheduler (CFS)? What are the pros and cons of setting each of them to very small and very large values?

**20.18** The Linux scheduler implements "soft" real-time scheduling. What features necessary for certain real-time programming tasks are missing? How might they be added to the kernel? What are the costs (downsides) of such features?

**20.19** Under what circumstances would a user process request an operation that results in the allocation of a demand-zero memory region?

**20.20** What scenarios would cause a page of memory to be mapped into a user program's address space with the copy-on-write attribute enabled?

**20.21** In Linux, shared libraries perform many operations central to the operating system. What is the advantage of keeping this functionality out of the kernel? Are there any drawbacks? Explain your answer.

**20.22** What are the benefits of a journaling file system such as Linux's ext3? What are the costs? Why does ext3 provide the option to journal only metadata?

**20.23** The directory structure of a Linux operating system could include files corresponding to several different file systems, including the Linux `/proc` file system. How might the need to support different file-system types affect the structure of the Linux kernel?

**20.24** In what ways does the Linux `setuid` feature differ from the `setuid` feature SVR4?

**20.25** The Linux source code is freely and widely available over the Internet and from CD-ROM vendors. What are three implications of this availability for the security of the Linux system?

CHAPTER 21

# Windows 10

**Updated by Alex Ionescu**

The Microsoft Windows 10 operating system is a preemptive multitasking client operating system for microprocessors implementing the Intel IA-32, AMD64, ARM, and ARM64 instruction set architectures (ISAs). Microsoft's corresponding server operating system, Windows Server 2016, is based on the same code as Windows 10 but supports only the 64-bit AMD64 ISAs. Windows 10 is the latest in a series of Microsoft operating systems based on its NT code, which replaced the earlier systems based on Windows 95/98. In this chapter, we discuss the key goals of Windows 10, the layered architecture of the system that has made it so easy to use, the file system, the networking features, and the programming interface.

## CHAPTER OBJECTIVES

- Explore the principles underlying Windows 10's design and the specific components of the system.
- Provide a detailed discussion of the Windows 10 file system.
- Illustrate the networking protocols supported in Windows 10.
- Describe the interface available in Windows 10 to system and application programmers.
- Describe the important algorithms implemented with Windows 10.

## 21.1 History

In the mid-1980s, Microsoft and IBM cooperated to develop the **OS/2 operating system**, which was written in assembly language for single-processor Intel 80286 systems. In 1988, Microsoft decided to end the joint effort with IBM and develop its own "new technology" (or NT) portable operating system to support both the OS/2 and POSIX application programming interfaces (APIs). In October 1988, Dave Cutler, the architect of the DEC VAX/VMS operating system, was hired and given the charter of building Microsoft's new operating system.

Originally, the team planned to use the OS/2 API as NT's native environment, but during development, NT was changed to use a new 32-bit Windows API (called Win32), based on the popular 16-bit API used in Windows 3.0. The first versions of NT were Windows NT 3.1 and Windows NT 3.1 Advanced Server. (At that time, 16-bit Windows was at Version 3.1.) Windows NT Version 4.0 adopted the Windows 95 user interface and incorporated Internet web-server and web-browser software. In addition, user-interface routines and all graphics code were moved into the kernel to improve performance (with the side effect of decreased system reliability and significant loss of security). Although previous versions of NT had been ported to other microprocessor architectures (including a brief 64-bit port to Alpha AXP 64), the Windows 2000 version, released in February 2000, supported only IA-32-compatible processors due to marketplace factors. Windows 2000 incorporated significant changes. It added Active Directory (an X.500-based directory service), better networking and laptop support, support for plug-and-play devices, a distributed file system, and support for more processors and more memory.

### 21.1.1 Windows XP, Vista, and 7

In October 2001, Windows XP was released as both an update to the Windows 2000 desktop operating system and a replacement for Windows 95/98. In April 2003, the server edition of Windows XP (called Windows Server 2003) became available. Windows XP updated the graphical user interface (GUI) with a visual design that took advantage of more recent hardware advances and many new ***ease-of-use features***. Numerous features were added to automatically repair problems in applications and the operating system itself. Because of these changes, Windows XP provided better networking and device experience (including zero-configuration wireless, instant messaging, streaming media, and digital photography/video). Windows Server 2003 provided dramatic performance improvements for large multiprocessors systems, as well as better reliability and security than earlier Windows operating systems.

The long-awaited update to Windows XP, called Windows Vista, was released in January 2007, but it was not well received. Although Windows Vista included many improvements that later continued into Windows 7, these improvements were overshadowed by Windows Vista's perceived sluggishness and compatibility problems. Microsoft responded to criticisms of Windows Vista by improving its engineering processes and working more closely with the makers of Windows hardware and applications.

The result was Windows 7, which was released in October 2009, along with corresponding server edition called Windows Server 2008 R2. Among the significant engineering changes was the increased use of **event tracing** rather than counters or profiling to analyze system behavior. Tracing runs constantly in the system, watching hundreds of scenarios execute. Scenarios include process startup and exit, file copy, and web-page load, for example. When one of these scenarios fails, or when it succeeds but does not perform well, the traces can be analyzed to determine the cause.

### 21.1.2 Windows 8

Three years later, in October 2012—amid an industry-wide pivot toward mobile computing and the world of **apps**—Microsoft released Windows 8, which represented the most significant change to the operating system since Windows XP. Windows 8 included a new user interface (named **Metro**) and a new programming model API (named **WinRT**). It also included a new way of managing applications (which ran under a new sandbox mechanism) through a **package system** that exclusively supported the new **Windows Store**, a competitor to the Apple App Store and the Android Store. Additionally, Windows 8 included a plethora of security, boot, and performance improvements. At the same time, support for "subsystems," a concept we'll describe further later in the chapter, was removed.

To support the new mobile world, Windows 8 was ported to the 32-bit ARM ISA for the first time and included multiple changes to the power management and hardware extensibility features of the kernel (discussed later in this chapter). Microsoft marketed two versions of this port. One version, called Windows RT, ran both Windows Store–packaged applications and some Microsoft-branded "classic" applications, such as Notepad, Internet Explorer, and most importantly, Office. The other version, called Windows Phone, could only run Windows Store–packaged applications.

For the first time ever, Microsoft released its own branded mobile hardware, under the "Surface" brand, which included the Surface RT, a tablet device that exclusively ran the Windows RT operating system. A bit later, Microsoft bought Nokia and began releasing Microsoft-branded phones as well, running Windows Phone.

Unfortunately, Windows 8 was a market failure, for several reasons. On the one hand, Metro focused on a tablet-oriented interface that forced users accustomed to older Windows operating systems to completely change the way they worked on their desktop computers. Windows 8, for example, replaced the start menu with touchscreen features, replaced shortcuts with animated "tiles," and offered little or no keyboard input support. On the other hand, the dearth of applications in the Windows Store, which was the only way to obtain apps for Microsoft's phone and tablet, led to the market failure of these devices as well, causing the company to eventually phase out the Surface RT device and write off the Nokia purchase.

Microsoft quickly sought to address many of these issues with the release of Windows 8.1 in October 2013. This release addressed many of the usability flaws of Windows 8 on nonmobile devices, bringing back more usability through a traditional keyboard and mouse, and provided ways to avoid the tile-based Metro interface. It also continued to improve on the many security, performance, and reliability changes introduced in Windows 8. Although this release was better received, the continued lack of applications in the Windows Store was a problem for the operating system's mobile market penetration, while desktop and server application programmers felt abandoned due to a lack of improvements in their area.

### 21.1.3 Windows 10

With the release of **Windows 10** in July 2015 and its server companion, Windows Server 2016, in October 2016, Microsoft shifted to a "Windows-as-a-Service" (WaaS) model (with included periodic functionality improvements). Windows 10 receives monthly incremental improvements called "feature rollups," as well as eight-month feature releases called "updates." Additionally, each upcoming release is made available to the public through the Windows Insider Program, or WIP, which releases versions on an almost weekly basis. Like cloud services and websites such as Facebook and Google, the new operating system uses live telemetry (sending debug information back to Microsoft) and tracing to dynamically enable and disable certain features for A/B testing (comparing how version "A" executes compared to similar version "B"), tries out new features while watching for compatibility issues, and aggressively adds or removes support for modern or legacy hardware. These dynamic configuration and testing features are what make this release an "as-a-service" implementation.

Windows 10 reintroduced the start menu, restored keyboard support, and deemphasized full-screen applications and live tiles. From the user′s perspective, these changes brought back the ease of use that users expected from Windows-based desktop operating systems. Additionally, Metro (which was renamed **Modern**) was redesigned so that Windows Store–packaged applications could be run on the regular desktop side by side with legacy applications. Finally, a new mechanism called the **Windows Desktop Bridge** made it possible to place Win32 applications in the Windows Store, mitigating the lack of applications written specifically for the newer systems. Meanwhile, Microsoft added support for C++11, C++14, and C++17 in the Visual Studio product, and many new APIs were added to the traditional Win32 programming API. A related change in Windows 10 was the release of the Unified Windows Platform (UWP) architecture, which allows applications to be written in such a way that they can execute on Windows for Desktop, Windows for IoT, XBOX One, Windows Phone, and Windows 10 Mixed Reality (previously known as Windows Holographic).

Windows 10 also replaced the concept of multiple subsystems, which had been removed in Windows 8 (as mentioned earlier), with a new mechanism called **Pico Providers**. This mechanism allows unmodified binaries belonging to a different operating system to run natively on Windows 10. In the "Anniversary Update" released in August 2016, this functionality was used to provide the Windows Subsystem for Linux, which can be used to run Linux ELF binaries in an entirely unmodified Ubuntu user-space environment.

In response to increased competitive pressures in the mobile and cloud-computing worlds, Microsoft also made power, performance, and scalability improvements in Windows 10, enabling it to run on a larger number of devices. In fact, a version called Windows 10 IoT Edition is specifically designed for environments such as the Raspberry Pi, while support for cloud-computing technologies such as containerization is built in through Docker for Windows. In Windows 10, the Microsoft Hyper-V virtualization technology is also built in, providing additional security and native support for running virtual machines. A special version of Windows Server, called Windows Server Nano, was also released. This extremely low-overhead server operating system is suited for containerized applications and other cloud-computing usages.

Windows 10 is a multiuser operating system, supporting simultaneous access through distributed services or through multiple instances of the GUI via Windows Terminal Services. The server editions of Windows 10 support simultaneous terminal server sessions from Windows desktop systems. The desktop editions of terminal server multiplex the keyboard, mouse, and monitor between virtual terminal sessions for each logged-on user. This feature, called ***fast user switching***, allows users to preempt each other at the console of a PC without having to log off and log on.

Let's return briefly to developments in the Windows GUI. We noted earlier that the GUI implementation moved into kernel mode in Windows NT 4.0 to improve performance. Further performance gains were made with the creation of a new user-mode component in Windows Vista, called the **Desktop Window Manager** (**DWM**). DWM provides the Windows interface look and feel on top of the Windows DirectX graphic software. DirectX continues to run in the kernel, as does the code (Win32k) implementing Windows' windowing and graphics model (User and GDI). Windows 7 made substantial changes to the DWM, significantly reducing its memory footprint and improving its performance, while Windows 10 made further improvements, especially in the areas of performance and security. Furthermore, Windows DirectX 11 and 12 include GPGPU mechanisms (general-purpose computing on GPU hardware) through **DirectCompute**, and many parts of Windows have been updated to take advantage of this high-performance graphics model. Through a new rendering layer called **CoreUI**, even legacy applications can now take advantage of DirectX-based rendering (creation of the final screen contents).

Windows XP was the first version of Windows to ship a 64-bit version (for the IA64 in 2003 and the AMD64 in 2005). Internally, the native NT file system (NTFS) and many of the Win32 APIs have always used 64-bit integers where appropriate. The major extension to 64-bit in Windows XP was meant as support for large virtual addresses. In addition, 64-bit editions of Windows support much larger physical memory, with the latest Windows Server 2016 release supporting up to 24 TB of RAM. By the time Windows 7 shipped, the AMD64 ISA had become available on almost all CPUs from both Intel and AMD. In addition, by that time, physical memory on client systems frequently exceeded the 4-GB limit of the IA-32. As a result, the 64-bit version of Windows 10 is now almost exclusively installed on client systems, apart from IoT and mobile systems. Because the AMD64 architecture supports high-fidelity IA-32 compatibility at the level of individual processes, 32- and 64-bit applications can be freely mixed in a single system. Interestingly, a similar pattern is now emerging on mobile systems. Apple iOS is the first mobile operating system to support the ARM64 architecture, which is the 64-bit ISA extension of ARM (also called AArch64). A future Windows 10 release will also officially ship with an ARM64 port designed for a new class of hardware, with compatibility for IA-32 architecture applications achieved through emulation and dynamic JIT recompilation.

In the rest of our description of Windows 10, we do not distinguish between the client editions and the corresponding server editions. They are based on the same core components and run the same binary files for the kernel and most drivers. Similarly, although Microsoft ships a variety of different editions of each release to address different market price points, few of the differences between editions are reflected in the core of the system. In this chapter, we focus primarily on the core components of Windows 10.

## 21.2 Design Principles

Microsoft's design goals for Windows included security, reliability, compatibility, high performance, extensibility, portability, and international support. Some additional goals, such as energy efficiency and dynamic device support, have recently been added to this list. Next, we discuss each of these goals and how each is achieved in Windows 10.

### 21.2.1 Security

Windows Vista and later security goals required more than just adherence to the design standards that had enabled Windows NT 4.0 to receive a C2 security classification from the U.S. government. (A C2 classification signifies a moderate level of protection from defective software and malicious attacks. Classifications were defined by the Department of Defense Trusted Computer System Evaluation Criteria, also known as the **Orange Book**.) Extensive code review and testing were combined with sophisticated automatic analysis tools to identify and investigate potential defects that might represent security vulnerabilities. Additionally, ***bug bounty*** participation programs allow external researchers and security professionals to identify, and submit, previously unknown security issues in Windows. In exchange, they receive monetary payment as well as credit in monthly security rollups, which are released by Microsoft to keep Windows 10 as secure as possible.

Windows traditionally based security on discretionary access controls. System objects, including files, registry keys, and kernel synchronization objects, are protected by **access-control lists** (**ACLs**) (see Section 13.4.2). ACLs are vulnerable to user and programmer errors, however, as well as to the most common attacks on consumer systems, in which the user is tricked into running code, often while browsing the Web. Windows Vista introduced a mechanism called **integrity levels** that acts as a rudimentary ***capability*** system for controlling access. Objects and processes are marked as having no, low, medium, or high system integrity. The integrity level determines what rights the objects and processes will have. For example, Windows does not allow a process to modify an object with a higher integrity level (based on its ***mandatory policy***), no matter what the setting of the ACL. Additionally, a process cannot read the memory of a higher-integrity process, no matter the ACL.

Windows 10 further strengthened the security model by introducing a combination of attribute-based access control (ABAC) and claim-based access control (CABC). Both features are used to implement dynamic access control (DAC) on server editions, as well as to support the capability-based system used by Windows Store applications and by Modern and packaged applications. With attributes and claims, system administrators need not rely on a user's name (or the group the user belongs to) as the only means that the security system can use to filter access to objects such as files. Properties of the user —such as, say, seniority in the organization, salary, and so on—can also be considered. These properties are encoded as ***attributes,*** which are paired with conditional access control entries in the ACL, such as "Seniority >= 10 Years."

Windows uses encryption as part of common protocols such as those used to communicate securely with websites. Encryption is also used to protect user files stored on secondary storage. Windows 7 and later versions allow users to easily encrypt entire volumes, as well as removable storage devices such as USB flash drives, with a feature called BitLocker. If a computer with an encrypted volume is stolen, the thieves will need very sophisticated technology (such as an electron microscope) to gain access to any of the computer's files, and it will be impossible for them to do so if the user has also configured an external USB-based token (unless the USB token was also stolen).

These types of security features focus on user and data security, but they are vulnerable to highly privileged programs that parse arbitrary content and that can be tricked due to programming errors into executing malicious code. Therefore, Windows also includes security measures often referred to as "exploit mitigations." These measures include wide-scope mitigations such as **address-space layout randomization** (**ASLR**), **Data Execution Prevention** (**DEP**), **Control-Flow Guard** (**CFG**), and **Arbitrary Code Guard** (**ACG**), as well as narrow-scope (targeted) mitigations specific to various exploitation techniques (which are outside the scope of this chapter).

Since 2001, chips from both Intel and AMD have allowed memory pages to be marked so that they cannot contain executable instruction code. The Windows DEP feature marks stacks and memory heaps (as well as all other data-only allocations) so that they cannot be used to execute code. This prevents attacks in which a program bug allows a buffer to overflow and then is tricked into executing the contents of the buffer. Additionally, starting with Windows 8.1, all kernel data-only memory allocations have been marked similarly.

Because DEP prevents attacker-controlled data from being executed as code, malicious developers moved on to **code reuse** attacks, in which existing executable code inside the program is reused in unexpected ways. (Only certain parts of the code are executed, and the flow is redirected from one instruction stream to another.) ASLR thwarts many forms of such attacks by randomizing the location of executable (and data) regions of memory, making it harder for code-reuse attacks to know where existing code is located. This safeguard makes it likely that a system under attack by a remote attacker will fail or crash.

No mitigation is perfect, however, and ASLR is no exception. For example, it may be ineffective against local attacks (in which some application is tricked into loading content from secondary storage, for example), as well as so-called **information leak** attacks (in which a program is tricked into revealing part of its address space). To address such problems, Windows 8.1 introduced a technology called CFG, which was much improved in Windows 10. CFG works with the compiler, the linker, the loader, and the memory manager to validate the destination address of any indirect branch (such as a call or jump) against a list of valid function prologues. If a program is tricked into redirecting control flow elsewhere through such an instruction, it crashes.

If attackers cannot bring executable data into an attack, nor reuse existing code, they may attempt to cause a program to allocate, on its own, executable and writeable code, which can then be filled by the attacker. Alternatively, the attackers might modify existing writeable data and mark it as executable data. Windows 10's ACG mitigation prohibits either of these operations. Once executable code is loaded, it can never be modified again, and once data is loaded, it can never be marked as executable.

Windows 10 has over thirty security mitigations in addition to those described here. This set of security features has made traditional attacks more difficult, perhaps explaining in part why crimeware applications, such as adware, credit card fraudware, and ransomware, have become so prevalent. These types of attacks rely on users to willingly and manually cause harm to their own computers (such as by double-clicking on applications against warning, or inputting their credit card number in a fake banking page). No operating system can be designed to militate against the gullibility and curiosity of human beings. Recently, Microsoft has started working directly with chip manufacturers, such as Intel, to build security mitigations directly into the ISA. One such mitigation, for example, is **Control-flow Enforcement Technology** (**CET**), which is a hardware implementation of CFG that also protects against return-oriented-programming (ROP) attacks by using hardware shadow stacks. A shadow stack contains the set of return addresses as stored when a routine is called. The addresses are checked for a mismatch before the return is executed. A mismatch means the stack has been compromised and action should be taken.

Another important aspect of security is integrity. Windows offers several **digital signature** facilities as part of its code integrity features. Windows uses digital signatures to ***sign*** operating system binaries so that it can verify that the files were produced by Microsoft or another known company. In non-IA-32 versions of Windows, the **code integrity** module is activated at boot to ensure that all the loaded modules in the kernel have valid signatures, assuring that they have not been tampered with. Additionally, ARM versions of Windows 8 extend the code integrity module with user-mode code integrity checks, which validate that all user programs have been signed by Microsoft or delivered through the Windows Store. A special version of Windows 10 (Windows 10 S, mostly meant for the education market) provides similar signing checks on all IA-32 and AMD64 systems. Digital signatures are also used as part of Code Integrity Guard, which allows applications to defend themselves against loading executable code from secondary storage that has not been appropriately signed. For example, an attacker might replace third-party binary with his own, but the digital signature would fail, and Code Integrity Guard would not load the binary into the processes' address space.

Finally, enterprise versions of Windows 10 make it possible to opt in to a new security feature called **Device Guard**. This mechanism allows organizations to customize the digital signing requirements of their computer systems, as well as blacklist and whitelist individual signing certificates or even binary hashes. For example, an organization could choose to allow only user-mode programs signed by Microsoft, Google, or Adobe to launch on their enterprise computers.

### 21.2.2 Reliability

Windows matured greatly as an operating system in its first ten years, leading to Windows 2000. At the same time, its reliability increased due to such factors as maturity in the source code, extensive stress testing of the system, improved CPU architectures, and automatic detection of many serious errors in drivers from both Microsoft and third parties. Windows has subsequently extended the tools for achieving reliability to include automatic analysis of source code for errors, tests to detect validation failures, and an application version of the driver verifier that applies dynamic checking for many common user-mode programming errors. Other improvements in reliability have resulted from moving more code out of the kernel and into user-mode services. Windows provides extensive support for writing drivers in user mode. System facilities that were once in the kernel and are now in user mode include the renderer for third-party fonts and much of the software stack for audio.

One of the most significant improvements in the Windows experience came from adding memory diagnostics as an option at boot time. This addition is especially valuable because so few consumer PCs have error-correcting memory. Bad RAM that lacks error correction and detection can change the data it stores—a change undetected by the hardware. The result is frustratingly erratic behavior in the system. The availability of memory diagnostics can warn users of a RAM problem. Windows 10 took this even further by introducing run-time memory diagnostics. If a machine encounters a kernel-mode crash more than five times in a row, and the crashes cannot be pinpointed to a specific cause or component, the kernel will use idle periods to move memory contents, flush system caches, and write repeated memory-testing patterns in all memory—all to preemptively discover if RAM is damaged. Users can then be informed of any issues without the need to reboot into the memory diagnostics tool at boot time.

Windows 7 also introduced a fault-tolerant memory heap. The heap learns from application crashes and automatically adjusts memory operations carried out by an application that has crashed. This makes the application more reliable even if it contains common bugs such as using memory after freeing it or accessing past the end of the allocation. Because such bugs can be exploited by attackers, Windows 7 also includes a mitigation for developers to block this feature and immediately crash any application with heap corruption. This is a very practical representation of the dichotomy that exists between the needs of security and the needs of user experience.

Achieving high reliability in Windows is particularly challenging because almost two billion systems run Windows. Even reliability problems that affect only a small percentage of these systems still impact tremendous numbers of users. The complexity of the Windows ecosystem also adds to the challenges. Millions of instances of applications, drivers, and other software are constantly being downloaded and run on Windows systems. Of course, there is also a constant stream of malware attacks. As Windows itself has become harder to attack directly, exploits increasingly target popular applications.

To cope with these challenges, Microsoft is increasingly relying on communications from customer machines to collect data from the ecosystem. Machines are sampled to see how they are performing, what software they are running, and what problems they are encountering. They automatically send data to Microsoft when their software, their drivers, or the kernel itself crashes or hangs. Features are measured to indicate how often they are used. Legacy behavior (methods no longer recommended for use by Microsoft) is sometimes disabled, and alerts are sent if attempts are made to use it again. The result is that Microsoft is building an ever-improving picture of what is happening in the Windows ecosystem that allows continuous improvements through software updates as well as providing data to guide future releases of Windows.

### 21.2.3 Windows and Application Compatibility

As mentioned, Windows XP was both an update of Windows 2000 and a replacement for Windows 95/98. Windows 2000 focused primarily on compatibility for business applications. The requirements for Windows XP included much higher compatibility with the consumer applications that ran on Windows 95/98. Application compatibility is difficult to achieve, for several reasons. For example, applications may check for a specific version of Windows, may depend to some extent on the quirks of the implementation of APIs, or may have latent application bugs that were masked in the previous system. Applications may also have been compiled for a different instruction set or have different expectations when run on today's multi-gigahertz, multicore systems. Windows 10 continues to focus on compatibility issues by implementing several strategies to run applications despite incompatibilities.

Like Windows XP, Windows 10 has a compatibility layer, called the shim engine, that sits between applications and the Win32 APIs. This engine can make Windows 10 look (almost) bug-for-bug compatible with previous versions of Windows. Windows 10 ships with a shim database of over 6,500 entries, describing particular quirks and tweaks that must be made for older applications. Furthermore, through the Application Compatibility Toolkit, users and administrators can build their own shim databases. Windows 10's **SwitchBranch** mechanism allows developers to choose which Windows version they'd like the Win32 API to emulate, including all the quirks and/or bugs of a previous API. The `Task Manager`'s “Operating System Context” column shows what SwitchBranch operating-system version each application is running under.

Windows 10, like earlier NT releases, maintains support for running many 16-bit applications using a ***thunking***, or conversion, layer—called Windows-on-Windows-32 (WoW32)—that translates 16-bit API calls into equivalent 32-bit calls. Similarly, the 64-bit version of Windows 10 provides a thunking layer, WoW64, that translates 32-bit API calls into native 64-bit calls. Finally, the ARM64 version of Windows 10 provides a dynamic JIT recompiler, translating IA-32 code, called WoWA64.

The original Windows subsystem model allows multiple operating-system personalities to be supported, as long as the applications are rebuilt as Portable Executable (PE) applications with a Microsoft compiler such as Visual Studio and source code is available. As noted earlier, although the API designed for Windows is the Win32API, some earlier editions of Windows supported a POSIX subsystem. POSIX is a standard specification for UNIX that allows UNIX-compatible software to be recompiled and run without modification on any POSIX-compatible operating system. Unfortunately, as Linux has matured, it has drifted farther and farther away from POSIX compatibility, and many modern Linux applications now rely on Linux-specific system calls and improvements to `glibc` that are not standardized. Additionally, it becomes impractical to ask users (or even enterprises) to recompile with Visual Studio every single Linux application that they'd like to use. Indeed, compiler differences among GCC, CLang, and Microsoft's C/C++ compiler often make doing so impossible. Therefore, even though the subsystem model still exists at an architectural level, the only subsystem on Windows going forward will be the Win32 subsystem itself, and compatibility with other operating systems is achieved through a new model that uses Pico Providers instead.

This significantly more powerful model extends the kernel via the ability to forward, or proxy, every system call, exception, fault, thread creation and termination, and process creation, along with a few other internal operations, to a secondary external driver (the Pico Provider itself). This secondary driver now becomes the owner of all such operations. While still using Windows 10's scheduler and memory manager (similar to a microkernel), it can implement its own ABI, system-call interface, executable file format parser, page fault handling, caching, I/O model, security model, and more.

Windows 10 includes one such Pico Provider, called LxCore, that is a multi-megabyte reimplementation of the Linux kernel. (Note that it is not Linux, and it does not share any code with Linux.) This driver is used by the “Windows Subsystem for Linux” feature, which can be used to load unmodified Linux ELF binaries without the need for source code or recompilation as PE binaries. Windows 10 users can run an unmodified Ubuntu user-mode file system (and, more recently, OpenSUSE and CentOS), servicing it with the `apt-get` package management command and running packages as normal. Note that the kernel reimplementation is not complete—many system calls are missing, as is access to most devices, since no Linux kernel drivers can load. Notably, while networking is fully supported, as well as serial devices, no GUI/frame-buffer access is possible.

As a final compatibility measure, Windows 8.1 and later versions also include the **Hyper-V for Client** feature. This allows applications to get bug-for-bug compatibility with Windows XP, Linux, and even DOS by running these operating systems inside a virtual machine.

### 21.2.4 Performance

Windows was designed to provide high performance on desktop systems (which are largely constrained by I/O performance), server systems (where the CPU is often the bottleneck), and large multithreaded and multiprocessor environments (where locking performance and cache-line management are keys to scalability). To satisfy performance requirements, NT used a variety of techniques, such as asynchronous I/O, optimized protocols for networks, kernel-based graphics rendering, and sophisticated caching of file-system data. The memory-management and synchronization algorithms were designed with an awareness of the performance considerations related to cache lines and multiprocessors.

Windows NT was designed for symmetrical multiprocessing (SMP); on a multiprocessor computer, several threads can run at the same time, even in the kernel. On each CPU, Windows NT uses priority-based preemptive scheduling of threads. Except while executing in the dispatcher or at interrupt level, threads in any process running in Windows can be preempted by higher-priority threads. Thus, the system responds quickly (see Chapter 5).

Windows XP further improved performance by reducing the code-path length in critical functions and implementing more scalable locking protocols, such as queued spinlocks and pushlocks. (**Pushlocks** are like optimized spinlocks with read–write lock features.) The new locking protocols helped reduce system bus cycles and included lock-free lists and queues, atomic read–modify–write operations (like `interlocked increment`), and other advanced synchronization techniques. These changes were needed because Windows XP added support for simultaneous multithreading (SMT), as well as a massively parallel pipelining technology that Intel had commercialized under the marketing name **Hyper Threading**. Because of this new technology, average home machines could appear to have two processors. A few years later, the introduction of multicore systems made multiprocessor systems the norm.

Next, Windows Server 2003, targeted toward large multiprocessor servers, was released, using even better algorithms and making a shift toward per-processor data structures, locks, and caches, as well as using page coloring and supporting NUMA machines. (Page coloring is a performance optimization to ensure that accesses to contiguous pages in virtual memory optimize use of the processor cache.) Windows XP 64-bit Edition was based on the Windows Server 2003 kernel so that early 64-bit adopters could take advantage of these improvements.

By the time Windows 7 was developed, several major changes had come to computing. The number of CPUs and the amount of physical memory available in the largest multiprocessors had increased substantially, so quite a lot of effort was put into further improving operating-system scalability.

The implementation of multiprocessing support in Windows NT used bitmasks to represent collections of processors and to identify, for example, which set of processors a particular thread could be scheduled on. These bitmasks were defined as fitting within a single word of memory, limiting the number of processors supported within a system to 64 on a 64-bit system and 32 on a 32-bit system. Thus, Windows 7 added the concept of **processor groups** to represent a collection of up to 64 processors. Multiple processor groups could be created, accommodating a total of more than 64 processors. Note that Windows calls a schedulable portion of a processor's execution unit a ***logical processor,*** as distinct from a physical processor or core. When we refer to a "processor" or "CPU" in this chapter, we really mean a "logical processor" from Windows's point of view. Windows 7 supported up to four processor groups, for a total of 256 logical processors, while Windows 10 now supports up to 20 groups, with a total of no more than 640 logical processors (therefore, not all groups can be fully filled).

All these additional CPUs created a great deal of contention for the locks used for scheduling CPUs and memory. Windows 7 broke these locks apart. For example, before Windows 7, a single lock was used by the Windows scheduler to synchronize access to the queues containing threads waiting for events. In Windows 7, each object has its own lock, allowing the queues to be accessed concurrently. Similarly, the global object manager lock, the cache manager VACB lock, and the memory manager PFN lock formerly synchronized access to large, global data structures. All were decomposed into more locks on smaller data structures. Also, many execution paths in the scheduler were rewritten to be lock-free. This change resulted in improved scalability performance for Windows 7 even on systems with 256 logical CPUs.

Other changes were due to the increasing importance of support for parallel computing. For years, the computer industry has been dominated by Moore's Law (see Section 1.1.3), leading to higher densities of transistors that manifest themselves as faster clock rates for each CPU. Moore's Law continues to hold true, but limits have been reached that prevent CPU clock rates from increasing further. Instead, transistors are being used to build more and more CPUs into each chip. New programming models for achieving parallel execution, such as Microsoft's Concurrency RunTime (ConcRT) and Parallel Processing Library (PPL), as well as Intel's Threading Building Blocks (TBB), are being used to express parallelism in C++ programs. Additionally, a vendor-neutral standard called OpenMP is supported by almost all compilers. Although Moore's Law has governed computing for forty years, it now seems that Amdahl's Law, which governs parallel computing (see Section 4.2), will rule the future.

Finally, power considerations have complicated design decisions around high-performance computing—especially in mobile systems, where battery life might trump performance needs, but also in cloud/server environments, where the cost of electricity might outweigh the need for the fastest possible computational result. Accordingly, Windows 10 now supports features that may sometimes sacrifice raw performance for better power efficiency. Examples include Core Parking, which puts an idle system into a sleep state, and Heterogeneous Multi Processing (HMP), which allocates tasks efficiently among cores.

To support task-based parallelism, the AMD64 ports of Windows 7 and later versions provide a new form of **user-mode scheduling** (**UMS**). UMS allows programs to be decomposed into tasks, and the tasks are then scheduled on the available CPUs by a scheduler that operates in user mode rather than in the kernel.

The advent of multiple CPUs on the smallest computers is only part of the shift taking place to parallel computing. Graphics processing units (GPUs) accelerate the computational algorithms needed for graphics by using **SIMD** architectures to execute a single instruction for multiple data at the same time. This has given rise to the use of GPUs for general computing, not just graphics. Operating-system support for software like OpenCL and CUDA is allowing programs to take advantage of the GPUs. Windows supports the use of GPUs through software in its DirectX graphics support. This software, called DirectCompute, allows programs to specify **computational kernels** using the "high-level shader language" programming model used by SIMD hardware. The computational kernels run very quickly on the GPU and return their results to the main computation running on the CPU. In Windows 10, the native graphics stack and many new Windows applications make use of DirectCompute, and new versions of Task Manager track GPU processor and memory usage, with DirectX now having its own GPU thread scheduler and GPU memory manager.

### 21.2.5 Extensibility

**Extensibility** refers to the capability of an operating system to keep up with advances in computing technology. To facilitate change over time, the developers implemented Windows using a layered architecture. The lowest-level kernel "executive" runs in kernel mode and provides the basic system services and abstractions that support shared use of the system. On top of the executive, several services operate in user mode. Among them were the environment subsystems that emulated different operating systems, which are deprecated today. Even in the kernel, Windows uses a layered architecture, with loadable drivers in the I/O system, so new file systems, new kinds of I/O devices, and new kinds of networking can be added while the system is running. Drivers

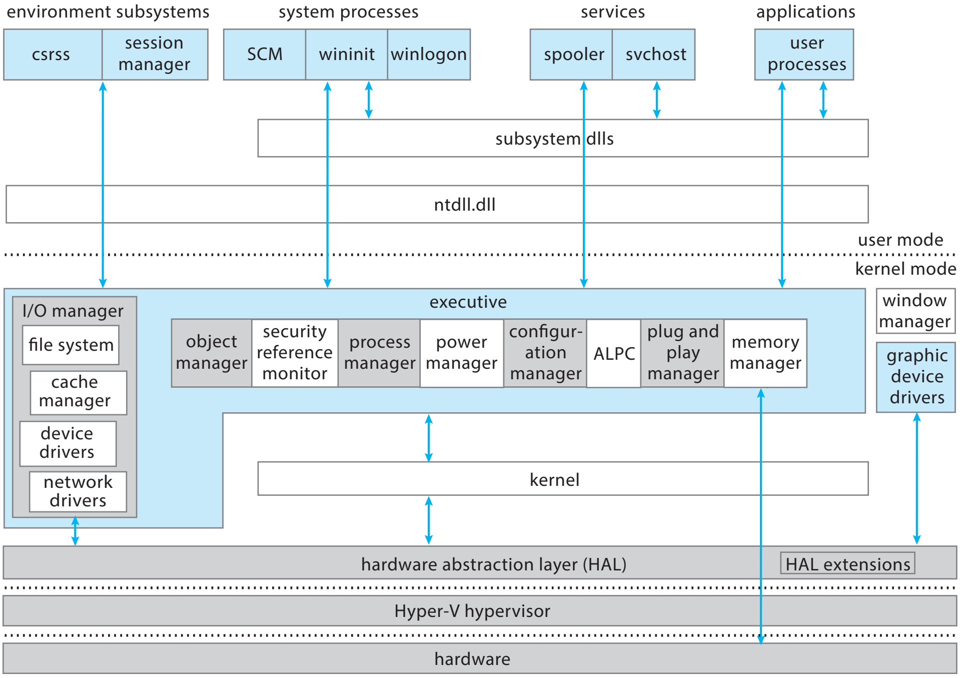

environment subsystems
system processes
services
applications
csrss
session manager
SCM
wininit
winlogon
spooler
svchost
user processes
subsystem dlls
ntdll.dll
user mode
kernel mode
executive
I/O manager
file system
cache manager
device drivers
network drivers
object manager
security reference monitor
process manager
power manager
configuration manager
ALPC
plug and play manager
memory manager
window manager
graphic device drivers
kernel
hardware abstraction layer (HAL)
HAL extensions
Hyper-V hypervisor
hardware

**Figure 21.1** Windows block diagram.

aren't limited to providing I/O functionality, however. As we've seen, a Pico Provider is also a type of loadable driver (as are most anti-malware drivers). Through Pico Providers and the modular structure of the system, additional operating system support can be added without affecting the executive. Figure Figure 21.1 shows the architecture of the Windows 10 kernel and subsystems.

Windows also uses a client–server model like the Mach operating system and supports distributed processing through **remote procedure calls** (**RPCs**) as defined by the Open Software Foundation. These RPCs take advantage of an executive component, called the **advanced local procedure call** (**ALPC**), that implements highly scalable communication between separate processes on a local machine. A combination of TCP/IP packets and named pipes over the SMB protocol is used for communication between processes across a network. On top of RPC, Windows implements the Distributed Common Object Model (DCOM) infrastructure, as well as the Windows Management Instrumentation (WMI) and Windows Remote Management (WinRM) mechanism, all of which can be used to rapidly extend the system with new services and management capabilities.

### 21.2.6 Portability

An operating system is **portable** if it can be moved from one CPU architecture to another with relatively few changes. Windows was designed to be portable. Like the UNIX operating system, Windows is written primarily in C and C++. There is relatively little architecture-specific source code and very little assembly code. Porting Windows to a new architecture mostly affects the Windows kernel, since the user-mode code in Windows is almost exclusively written to be architecture independent. To port Windows, the kernel's architecture-specific code must be rewritten for the target CPU, and sometimes conditional compilation is needed in other parts of the kernel because of changes in major data structures, such as the page-table format. The entire Windows system must then be recompiled for the new CPU instruction set.

Operating systems are sensitive not only to CPU architecture but also to CPU support chips and hardware boot programs. The CPU and support chips are collectively known as the **chipset**. These chipsets and the associated boot code determine how interrupts are delivered, describe the physical characteristics of each system, and provide interfaces to deeper aspects of the CPU architecture, such as error recovery and power management. It would be burdensome to have to port Windows to each type of support chip as well as to each CPU architecture. Instead, Windows isolates most of the chipset-dependent code in a dynamic link library (DLL), called the **hardware-abstraction layer** (**HAL**), that is loaded with the kernel.

The Windows kernel depends on the HAL interfaces rather than on the underlying chipset details. This allows the single set of a kernel and driver binaries for a particular CPU to be used with different chipsets simply by loading a different version of the HAL. Originally, to support the many architectures that Windows ran on, and the many computer companies and designs in the market, over 450 different HALs existed. Over time, the advent of standards such as the Advanced Configuration and Power Interface (ACPI), the increasing similarity of components available in the marketplace, and the merging of computer manufacturers led to changes; today, the AMD64 port of Windows 10 comes with a single HAL. Interestingly, though, no such developments have yet occurred in the market for mobile devices. Today, Windows supports a limited number of ARM chipsets—and must have the appropriate HAL code for each of them. To avoid going back to a model of multiple HALs, Windows 8 introduced the concept of HAL Extensions, which are DLLs that are loaded dynamically by the HAL based on the detected SoC (system on a chip) components, such as the interrupt controller, timer manager, and DMA controller.

Over the years, Windows has been ported to a number of different CPU architectures: Intel IA-32-compatible 32-bit CPUs, AMD64-compatible and IA64 64-bit CPUs, and DEC Alpha, DEC Alpha AXP64, MIPS, and PowerPC CPUs. Most of these CPU architectures failed in the consumer desktop market. When Windows 7 shipped, only the IA-32 and AMD64 architectures were supported on client computers, along with AMD64 on servers. With Windows 8, 32-bit ARM was added, and Windows 10 now supports ARM64 as well.

### 21.2.7 International Support

Windows was designed for international and multinational use. It provides support for different locales via the **national-language-support** (**NLS**) API. The NLS API provides specialized routines to format dates, time, and money in accordance with national customs. String comparisons are specialized to account for varying character sets. UNICODE is Windows's native character code, specifically in its UTL-16LE encoding format (which is different from

Linux's and the Web's standard UTF-8). Windows supports ANSI characters by converting them to UNICODE characters before manipulating them (8-bit to 16-bit conversion).

System text strings are kept in resource tables inside files that can be replaced to localize the system for different languages. Before Windows Vista, Microsoft shipped these resource tables inside the DLLs themselves, which meant that different executable binaries existed for each different version of Windows and only one language was available at a single time. With Windows Vista's **multiple user interface** (**MUI**) support, multiple locales can be used concurrently, which is important to multilingual individuals and businesses. This was achieved by moving all of the resource tables into separate `.mui` files that live in the appropriate language directory alongside the `.dll` file, with support in the loader to pick the appropriate file based on the currently selected language.

### 21.2.8 Energy Efficiency

Increasing energy efficiency causes batteries to last longer for laptops and Internet-only netbooks, saves significant operating costs for power and cooling of data centers, and contributes to green initiatives aimed at lowering energy consumption by businesses and consumers. For some time, Windows has implemented several strategies for decreasing energy use. The CPUs are moved to lower power states—for example, by lowering clock frequency—whenever possible. In addition, when a computer is not being actively used, Windows may put the entire computer into a low-power state (sleep) or may even save all of memory to secondary storage and shut the computer off (hibernation). When the user returns, the computer powers up and continues from its previous state, so the user does not need to reboot and restart applications.

The longer a CPU can stay unused, the more energy can be saved. Because computers are so much faster than human beings, a lot of energy can be saved just while humans are thinking. The problem is that many programs are polled to wait for activity, and software timers are frequently expiring, keeping the CPU from staying idle long enough to save much energy.

Windows 7 extends CPU idle time by delivering clock-tick interrupts only to logical CPU 0 and all other currently active CPUs (skipping idle ones) and by coalescing eligible software timers into smaller numbers of events. On server systems, it also "parks" entire CPUs when systems are not heavily loaded. Additionally, timer expiration is not distributed, and a single CPU is typically in charge of handling all software timer expirations. A thread that was running on, say, logical CPU 3 does not cause CPU 3 to wake up and service this expiration if it is currently idle when another, nonsleeping CPU could handle it instead.

While these measures helped, they were not enough to increase battery life in mobile systems such as phones, which have a fraction of the battery capacity of laptops. Windows 8 thus introduced a number of features to further optimize battery life. First, the WinRT programming model does not allow for precise timers with a guaranteed expiration time. All timers registered through the new API are candidates for coalescing, unlike Win32 timers, which had to be manually opted in. Next, the concept of a **dynamic tick** was introduced, in which CPU0 is no longer the **clock owner**, and the last-active CPU takes on this responsibility.

More significantly, the entire Metro/Modern/UWP application model delivered through the Windows Store includes a feature, the **Process Lifetime Manager** (**PLM**), that automatically suspends all of the threads in a process that has been idle for more than a few seconds. This not only mitigates the constant polling behavior of many applications, but also removes the ability for UWP applications to do their own background work (such as querying the GPS location), forcing them to deal with a system of **brokers** that efficiently coalesce audio, location, download, and other requests and can cache data while the process is suspended.

Finally, using a new component called the **Desktop Activity Moderator** (**DAM**), Windows 8 and later versions support a new type of system state called **Connected Standby**. Imagine putting a computer to sleep—this action takes several seconds, after which everything on the computer appears to disappear, with all the hardware turning off. Pressing a button on the keyboard wakes up the computer, which takes a few additional seconds, and everything resumes. On a phone or tablet, however, putting the device to sleep is not expected to take seconds—users want their screen to turn off immediately. But if Windows merely turned off the screen, all programs would continue running, and legacy Win32 applications, lacking a PLM and timer coalescing, would continue to poll, perhaps even waking up the screen again. Battery life would drain significantly.

Connected Standby addresses this problem by virtually freezing the computer when the power button is pressed or the screen turns off—without really putting the computer to sleep. The hardware clock is stopped, all processes and services are suspended, and all timer expirations are delayed 30 minutes. The net effect, even though the computer is still running, is that it runs in such a almost-total state of idleness that the processor and peripherals can effectively run in their lowest power state. Special hardware and firmware are required to fully support this mode; for example, the Surface-branded tablet hardware includes this capability.

### 21.2.9 Dynamic Device Support

Early in the history of the PC industry, computer configurations were fairly static, although new devices might occasionally be plugged into the serial, printer, or game ports on the back of a computer. The next steps toward dynamic configuration of PCs were laptop docks and PCMCIA cards. Using such a device, a PC could quickly be connected to or disconnected from a full set of peripherals. Contemporary PCs are designed to enable users to plug and unplug a huge host of peripherals frequently.

Support for dynamic configuration of devices is continually evolving in Windows. The system can automatically recognize devices when they are plugged in and can find, install, and load the appropriate drivers—often without user intervention. When devices are unplugged, the drivers automatically unload, and system execution continues without disrupting other software. Additionally, Windows Update permits downloading of third-party drivers directly through Microsoft, avoiding the usage of installation DVDs or having the user scour the manufacturer's website.

Beyond peripherals, Windows Server also supports dynamic hot-add and hot-replace of CPUs and RAM, as well as dynamic hot-remove of RAM. These features allow the components to be added, replaced, or removed without system interruption. While of limited use in physical servers, this technology is key to dynamic scalability in cloud computing, especially in Infrastructure-as-a-Service (IaaS) and cloud computing environments. In these scenarios, a physical machine can be configured to support a limited number of its processors based on a service fee, which can then be dynamically upgraded, without requiring a reboot, through a compatible hypervisor such as Hyper-V and a simple slider in the owner's user interface.

## 21.3 System Components

The architecture of Windows is a layered system of modules operating at specific privilege levels, as shown earlier in Figure 21.1. By default, these privilege levels are first implemented by the processor (providing a "vertical" privilege isolation between user mode and kernel mode). Windows 10 can also use its Hyper-V hypervisor to provide an orthogonal (logically independent) security model through **Virtual Trust Levels** (**VTLs**). When users enable this feature, the system operates in a Virtual Secure Mode (VSM). In this mode, the layered privileged system now has two implementations, one called the **Normal World**, or VTL 0, and one called the **Secure World**, or VTL 1. Within each of these worlds, we find a user mode and a kernel mode.

Let's look at this structure in somewhat more detail.

- In the Normal World, in kernel mode are (1) the HAL and its extensions and (2) the kernel and its executive, which load drivers and DLL dependencies. In user mode are a collection of system processes, the Win32 environment subsystem, and various services.
- In the Secure World, if VSM is enabled, are a secure kernel and executive (within which a secure micro-HAL is embedded). A collection of isolated **Trustlets** (discussed later) run in secure user mode.
- Finally, the bottommost layer in Secure World runs in a special processor mode (called, for example, VMX Root Mode on Intel processors), which contains the Hyper-V hypervisor component, which uses hardware virtualization to construct the Normal-to-Secure-World boundary. (The user-to-kernel boundary is provided by the CPU natively.)

One of the chief advantages of this type of architecture is that interactions between modules, and between privilege levels, are kept simple, and that isolation needs and security needs are not necessarily conflated through privilege. For example, a secure, protected component that stores passwords can itself be unprivileged. In the past, operating-system designers chose to meet isolation needs by making the secure component highly privileged, but this results in a net loss for the security of the system when this component is compromised.

The remainder of this section describes these layers and subsystems.

### 21.3.1 Hyper-V Hypervisor

The hypervisor is the first component initialized on a system with VSM enabled, which happens as soon as the user enables the Hyper-V component. It is used both to provide hardware virtualization features for running separate virtual machines and to provide the VTL boundary and related access to the hardware's Second Level Address Translation (SLAT) functionality (discussed shortly). The hypervisor uses a CPU-specific virtualization extension, such as AMD's Pacifica (SVMX) or Intel's Vanderpool (VT-x), to intercept any interrupt, exception, memory access, instruction, port, or register access that it chooses and deny, modify, or redirect the effect, source, or destination of the operation. It also provides a **hypercall** interface, which enables it to communicate with the kernel in VTL 0, the secure kernel in VTL 1, and all other running virtual machine kernels and secure kernels.

### 21.3.2 Secure Kernel

The secure kernel acts as the kernel-mode environment of isolated (VTL 1) user-mode Trustlet applications (applications that implement parts of the Windows security model). It provides the same system-call interface that the kernel does, so that all interrupts, exceptions, and attempts to enter kernel mode from a VTL 1 Trustlet result in entering the secure kernel instead. However, the secure kernel is not involved in context switching, thread scheduling, memory management, interprocess-communication, or any of the other standard kernel tasks. Additionally, no kernel-mode drivers are present in VTL 1. In an attempt to reduce the attack surface of the Secure World, these complex implementations remain the responsibility of Normal World components. Thus, the secure kernel acts as a type of "proxy kernel" that hands off the management of its resources, paging, scheduling, and more, to the regular kernel services in VTL 0. This does make the Secure World vulnerable to denial-of-service attacks, but that is a reasonable tradeoff of the security design, which values data privacy and integrity over service guarantees.

In addition to forwarding system calls, the secure kernel's other responsibility is providing access to the hardware secrets, the trusted platform module (TPM), and code integrity policies that were captured at boot. With this information, Trustlets can encrypt and decrypt data with keys that the Normal World cannot obtain and can sign and attest (co-sign by Microsoft) reports with integrity tokens that cannot be faked or replicated outside of the Secure World. Using a CPU feature called Second Level Address Translation (SLAT), the secure kernel also provides the ability to allocate virtual memory in such a way that the physical pages backing it cannot be seen at all from the Normal World. Windows 10 uses these capabilities to provide additional protection of enterprise credentials through a feature called Credential Guard.

Furthermore, when Device Guard (mentioned earlier) is activated, it takes advantage of VTL 1 capabilities by moving all digital signature checking into the secure kernel. This means that even if attacked through a software vulnerability, the normal kernel cannot be forced to load unsigned drivers, as the VTL 1 boundary would have to be breached for that to occur. On a Device Guard–protected system, for a kernel-mode page in VTL 0 to be authorized for execution, the kernel must first ask permission from the secure kernel, and only the secure kernel can grant this page executable access. More secure deployments (such as in embedded or high-risk systems) can require this level of signature validation for user-mode pages as well.

Additionally, work is being done to allow special classes of hardware devices, such as USB webcams and smartcard readers, to be directly managed by user-mode drivers running in VTL 1 (using the UMDF framework described later), allowing biometric data to be securely captured in VTL 1 without any component in the Normal World being able to intercept it. Currently, the only Trustlets allowed are those that provide the Microsoft-signed implementation of Credential Guard and virtual-TPM support. Newer versions of Windows 10 will also support **VSM Enclaves**, which will allow validly signed (but not necessarily Microsoft-signed) third-party code wishing to perform its own cryptographic calculations to do so. Software enclaves will allow regular VTL 0 applications to "call into" an enclave, which will run executable code on top of input data and return presumably encrypted output data.

For more information on the secure kernel, see https://blogs.technet.microsoft.com/ash/2016/03/02/windows-10-device-guard-and-credential-guard-demystified/.

### 21.3.3 Hardware-Abstraction Layer

The HAL is the layer of software that hides hardware chipset differences from upper levels of the operating system. The HAL exports a virtual hardware interface that is used by the kernel dispatcher, the executive, and the device drivers. Only a single version of each device driver is required for each CPU architecture, no matter what support chips might be present. Device drivers map devices and access them directly, but the chipset-specific details of mapping memory, configuring I/O buses, setting up DMA, and coping with motherboard-specific facilities are all provided by the HAL interfaces.

### 21.3.4 Kernel

The kernel layer of Windows has the following main responsibilities: thread scheduling and context switching, low-level processor synchronization, interrupt and exception handling, and switching between user mode and kernel mode through the system-call interface. Additionally, the kernel layer implements the initial code that takes over from the boot loader, formalizing the transition into the Windows operating system. It also implements the initial code that safely crashes the kernel in case of an unexpected exception, assertion, or other inconsistency. The kernel is mostly implemented in the C language, using assembly language only when absolutely necessary to interface with the lowest level of the hardware architecture and when direct register access is needed.

#### 21.3.4.1 Dispatcher

The dispatcher provides the foundation for the executive and the subsystems. Most of the dispatcher is never paged out of memory, and its execution is never preempted. Its main responsibilities are thread scheduling and context switching, implementation of synchronization primitives, timer management, software interrupts (asynchronous and deferred procedure calls), interprocessor interrupts (IPIs) and exception dispatching. It also manages hardware and software interrupt prioritization under the system of **interrupt request levels** (**IRQLs**).

#### 21.3.4.2 Switching Between User-Mode and Kernel-Mode Threads

What the programmer thinks of as a thread in traditional Windows is actually a thread with two modes of execution: a **user-mode thread** (**UT**) and a **kernel-mode thread** (**KT**). The thread has two stacks, one for UT execution and the other for KT. A UT requests a system service by executing an instruction that causes a trap to kernel mode. The kernel layer runs a trap handler that switches UT stack to its KT sister and changes CPU mode to kernel. When thread in KT mode has completed its kernel execution and is ready to switch back to the corresponding UT, the kernel layer is called to make the switch to the UT, which continues its execution in user mode. The KT switch also happens when an interrupt occurs.

Windows 7 modifies the behavior of the kernel layer to support user-mode scheduling of the UTs. User-mode schedulers in Windows 7 support cooperative scheduling. A UT can explicitly yield to another UT by calling the user-mode scheduler; it is not necessary to enter the kernel. User-mode scheduling is explained in more detail in Section 21.7.3.7.

In Windows, the dispatcher is not a separate thread running in the kernel. Rather, the dispatcher code is executed by the KT component of a UT thread. A thread goes into kernel mode in the same circumstances that, in other operating systems, cause a kernel thread to be called. These same circumstances will cause the KT to run through the dispatcher code after its other operations, determining which thread to run next on the current core.

#### 21.3.4.3 Threads

Like many other modern operating systems, Windows uses threads as the key schedulable unit of executable code, with processes serving as containers of threads. Therefore, each process must have at least one thread, and each thread has its own scheduling state, including actual priority, processor affinity, and CPU usage information.

There are eight possible thread states: `initializing`, `ready`, `deferred-ready`, `standby`, `running`, `waiting`, `transition`, and `terminated`. `ready` indicates that the thread is waiting to execute, while `deferred-ready` indicates that the thread has been selected to run on a specific processor but has not yet been scheduled. A thread is `running` when it is executing on a processor core. It runs until it is preempted by a higher-priority thread, until it terminates, until its allotted execution time (quantum) ends, or until it waits on a dispatcher object, such as an event signaling I/O completion. If a thread is preempting another thread on a different processor, it is placed in the `standby` state on that processor, which means it is the next thread to run.

Preemption is instantaneous—the current thread does not get a chance to finish its quantum. Therefore, the processor sends a software interrupt—in this case, a **deferred procedure call** (**DPC**)—to signal to the other processor that a thread is in the `standby` state and should be immediately picked up for execution. Interestingly, a thread in the `standby` state can itself be preempted if yet another processor finds an even higher-priority thread to run in this processor. At that point, the new higher-priority thread will go to `standby`, and the previous thread will go to the `ready` state. A thread is in the `waiting` state when it is waiting for a dispatcher object to be signaled. A thread is in the `transition` state while it waits for resources necessary for execution; for example, it may be waiting for its kernel stack to be paged in from secondary storage. A thread enters the `terminated` state when it finishes execution, and a thread begins in the `initializing` state as it is being created, before becoming `ready` for the first time.

The dispatcher uses a 32-level priority scheme to determine the order of thread execution. Priorities are divided into two classes: variable class and static class. The variable class contains threads having priorities from 1 to 15, and the static class contains threads with priorities ranging from 16 to 31. The dispatcher uses a linked list for each scheduling priority; this set of lists is called the **dispatcher database**. The database uses a bitmap to indicate the presence of at least one entry in the list associated with the priority of the bit's position. Therefore, instead of having to traverse the set of lists from highest to lowest until it finds a thread that is ready to run, the dispatcher can simply find the list associated with the highest bit set.

Prior to Windows Server 2003, the dispatcher database was global, resulting in heavy contention on large CPU systems. In Windows Server 2003 and later versions, the global database was broken apart into per-processor databases, with per-processor locks. With this new model, a thread will only be in the database of its **ideal processor**. It is thus guaranteed to have a processor affinity that includes the processor on whose database it is located. The dispatcher can now simply pick the first thread in the list associated with the highest bit set and does not have to acquire a global lock. Dispatching is therefore a constant-time operation, parallelizable across all CPUs on the machine.

On a single-processor system, if no ready thread is found, the dispatcher executes a special thread called the ***idle thread,*** whose role is to begin the transition to one of the CPU's initial sleep states. Priority class 0 is reserved for the idle thread. On a multiprocessor system, before executing the idle thread, the dispatcher looks at the dispatcher databases of other nearby processors, taking caching topologies and NUMA node distances into consideration. This operation requires acquiring the locks of other processor cores in order to safely inspect their lists. If no thread can be stolen from a nearby core, the dispatcher looks at the next nearest core, and so on. If no threads can be stolen at all, then the processor executes the idle thread. Therefore, in a multiprocessor system, each CPU will have its own idle thread.

Putting each thread on only the dispatcher database of its ideal processor causes a locality problem. Imagine a CPU executing a thread at priority 2 in a CPU-bound way, while another CPU is executing a thread at priority 18, also CPU-bound. Then, a thread at priority 17 becomes `ready`. If the ideal processor of this thread is the first CPU, the thread preempts the current `running` thread. But if the ideal processor is the latter CPU, it goes into the `ready` queue instead, waiting for its turn to run (which won't happen until the priority 17 thread gives up the CPU by terminating or entering a wait state).

Windows 7 introduced a load-balancer algorithm to address this situation, but it was a heavy-handed and disruptive approach to the locality issue. Windows 8 and later versions solved the problem in a more nuanced way. Instead of a global database as in Windows XP and earlier versions, or a per-processor database as in Windows Server 2003 and later versions, the newer Windows versions combine these approaches to form a **shared ready queue** among a group of some, but not all, processors. The number of CPUs that form one shared group depends on the topology of the system, as well as on whether it is a server or client system. The number is chosen to keep contention low on very large processor systems, while avoiding locality (and thus latency and contention) issues on smaller client systems. Additionally, processor affinities are still respected, so that a processor in a given group is guaranteed that all threads in the shared ready queue are appropriate—it never needs to “skip” over a thread, keeping the algorithm constant time.

Windows has a timer expire every 15 milliseconds to create a clock “tick” to examine system states, update the time, and do other housekeeping. That tick is received by the thread on every non-idle core. The interrupt handler (being run by the thread, now in KT mode) determines if the thread’s quantum has expired. When a thread’s time quantum runs out, the clock interrupt queues a quantum-end DPC to the processor. Queuing the DPC results in a software interrupt when the processor returns to normal interrupt priority. The software interrupt causes the thread to run dispatcher code in KT mode to reschedule the processor to execute the next `ready` thread at the preempted thread’s priority level in a round-robin fashion. If no other thread at this level is `ready`, a lower-priority `ready` thread is not chosen, because a higher-priority `ready` thread already exists—the one that exhausted its quantum in the first place. In this situation, the quantum is simply restored to its default value, and the same thread executes once again. Therefore, Windows always executes the highest-priority `ready` thread.

When a variable-priority thread is awakened from a wait operation, the dispatcher may boost its priority. The amount of the boost depends on the type of wait associated with the thread. If the wait was due to I/O, then the boost depends on the device for which the thread was waiting. For example, a thread waiting for sound I/O would get a large priority increase, whereas a thread waiting for a disk operation would get a moderate one. This strategy enables I/O-bound threads to keep the I/O devices busy while permitting compute-bound threads to use spare CPU cycles in the background.

Another type of boost is applied to threads waiting on mutex, semaphore, or event synchronization objects. This boost is usually a hard-coded value of one priority level, although kernel drivers have the option of making a different change. (For example, the kernel-mode GUI code applies a boost of two priority levels to all GUI threads waking up to process window messages.) This strategy is used to reduce the latency between when a lock or other notification mechanism is signaled and when the next waiter in line executes in response to the state change.

In addition, the thread associated with the user’s active GUI window receives a priority boost of two whenever it wakes up for any reason, on top of any other existing boost, to enhance its response time. This strategy, called the ***foreground priority separation boost,*** tends to give good response times to interactive threads.

Finally, Windows Server 2003 added a lock-handoff boost for certain classes of locks, such as critical sections. This boost is similar to the mutex, semaphore, and event boost, except that it tracks ownership. Instead of boosting the waking thread by a hard-coded value of one priority level, it boosts to one priority level above that of the current owner (the one releasing the lock). This helps in situations where, for example, a thread at priority 12 is releasing a mutex, but the waiting thread is at priority 8. If the waiting thread receives a boost only to 9, it will not be able to preempt the releasing thread. But if it receives a boost to 13, it can preempt and instantly acquire the critical section.

Because threads may run with boosted priorities when they wake up from waits, the priority of a thread is lowered at the end of every quantum as long as the thread is above its base (initial) priority. This is done according to the following rule: For I/O threads and threads boosted due to waking up because of an event, mutex, or semaphore, one priority level is lost at quantum end. For threads boosted due to the lock-handoff boost or the foreground priority separation boost, the entire value of the boost is lost. Threads that have received boosts of both types will obey both of these rules (losing one level of the first boost, as well as the entirety of the second boost). Lowering the thread's priority makes sure that the boost is applied only for latency reduction and for keeping I/O devices busy, not to give undue execution preference to compute-bound threads.

#### 21.3.4.4 Thread Scheduling

Scheduling occurs when a thread enters the `ready` or `waiting` state, when a thread terminates, or when an application changes a thread's processor affinity. As we have seen throughout the text, a thread could become ready at any time. If a higher-priority thread becomes ready while a lower-priority thread is running, the lower-priority thread is preempted immediately. This preemption gives the higher-priority thread instant access to the CPU, without waiting on the lower-priority thread's quantum to complete.

It is the lower-priority thread itself, performing some event that caused it to operate in the dispatcher, that wakes up the waiting thread and immediately context-switches to it while placing itself back in the `ready` state. This model essentially distributes the scheduling logic throughout dozens of Windows kernel functions and makes each currently running thread behave as the scheduling entity. In contrast, other operating systems rely on an external "scheduler thread" triggered periodically based on a timer. The advantage of the Windows approach is latency reduction, with the cost of added overhead inside every I/O and other state-changing operation, which causes the current thread to perform scheduler work.

Windows is not a hard-real-time operating system, however, because it does not guarantee that any thread, even the highest-priority one, will start to execute within a particular time limit or have a guaranteed period of execution. Threads are blocked indefinitely while DPCs and **interrupt service routines** (**ISRs**) are running (as further discussed below), and they can be preempted at any time by a higher-priority thread or be forced to round-robin with another thread of equal priority at quantum end.

Traditionally, the Windows scheduler uses sampling to measure CPU utilization by threads. The system timer fires periodically, and the timer interrupt handler takes note of what thread is currently scheduled and whether it is executing in user or kernel mode when the interrupt occurred. This sampling technique originally came about because either the CPU did not have a high-resolution clock or the clock was too expensive or unreliable to access frequently. Although efficient, sampling is inaccurate and leads to anomalies such as charging the entire duration of the clock (15 milliseconds) to the currently running thread (or DPC or ISR). Therefore, the system ends up completely ignoring some number of milliseconds—say, 14.999—that could have been spent idle, running other threads, running other DPCs and ISRs, or a combination of all of these operations. Additionally, because quantum is measured based on clock ticks, this causes the premature round-robin selection of a new thread, even though the current thread may have run for only a fraction of the quantum.

Starting with Windows Vista, execution time is also tracked using the hardware **timestamp counter** (**TSC**) included in all processors since the Pentium Pro. Using the TSC results in more accurate accounting of CPU usage (for applications that use it—note that Task Manager does not) and also causes the scheduler not to switch out threads before they have run for a full quantum. Additionally, Windows 7 and later versions track, and charge, the TSC to ISRsand DPCs, resulting in more accurate "Interrupt Time" measurements as well (again, for tools that use this new measurement). Because all possible execution time is now accounted for, it is possible to add it to idle time (which is also tracked using the TSC) and accurately compute the exact number of CPU cycles out of all possible CPU cycles in a given period (due to the fact that modern processors have dynamically shifting frequencies), resulting in cycle-accurate CPU usage measurements. Tools such as Microsoft's SysInternals Process Explorer use this mechanism in their user interface.

#### 21.3.4.5 Implementation of Synchronization Primitives

Windows uses a number of **dispatcher objects** to control dispatching and synchronization in the system. Examples of these objects include the following:

- The **event** is used to record an event occurrence and to synchronize this occurrence with some action. Notification events signal all waiting threads, and synchronization events signal a single waiting thread.
- The **mutex** provides kernel-mode or user-mode mutual exclusion associated with the notion of ownership.
- The **semaphore** acts as a counter or gate to control the number of threads that access a resource.
- The **thread** is the entity that is scheduled by the kernel dispatcher. It is associated with a process, which encapsulates a virtual address space, list of open resources, and more. The thread is signaled when the thread exits, and the process, when the process exits (that is, when all of its threads have exited).
- The **timer** is used to keep track of time and to signal timeouts when operations take too long and need to be interrupted or when a periodic activity needs to be scheduled. Just like events, timers can operate in notification mode (signal all) or synchronization mode (signal one).

All of the dispatcher objects can be accessed from user mode via an `open` operation that returns a handle. The user-mode code waits on handles to synchronize with other threads as well as with the operating system (see Section 21.7.1).

#### 21.3.4.6 Interrupt Request Levels (IRQLs)

Both hardware and software interrupts are prioritized and are serviced in priority order. There are 16 interrupt request levels (IRQLs) on all Windows ISAs except the legacy IA-32, which uses 32. The lowest level, IRQL 0, is called the PASSIVE_LEVEL and is the default level at which all threads execute, whether in kernel or user mode. The next levels are the software interrupt levels for APCs and DPCs. Levels 3 to 10 are used to represent hardware interrupts based on selections made by the PnP manager with the help of the HAL and the PCI/ACPI bus drivers. Finally, the uppermost levels are reserved for the clock interrupt (used for quantum management) and IPI delivery. The last level, HIGH_LEVEL, blocks all maskable interrupts and is typically used when crashing the system in a controlled manner.

The Windows IRQLs are defined in Figure 21.2.

#### 21.3.4.7 Software Interrupts: Asynchronous and Deferred Procedure Calls

The dispatcher implements two types of software interrupts: **asynchronous procedure calls** (**APCs**) and deferred procedure calls (DPCs, mentioned earlier). APCs are used to suspend or resume existing threads, terminate threads, deliver notifications that an asynchronous I/O has completed, and extract or modify the contents of the CPU registers (the context) from a running thread. APCs are queued to specific threads and allow the system to execute both system and user code within a process's context. User-mode execution of an APC cannot occur at arbitrary times, but only when the thread is waiting and is marked ***alertable.*** Kernel-mode execution of an APC, in contrast, instantaneously executes in the context of a running thread because it is delivered as a software interrupt running at IRQL 1 (APC_LEVEL), which is higher than the default IRQL 0 (PASSIVE_LEVEL). Additionally, even if a thread is waiting in kernel mode, the wait can be broken by the APC and resumed once the APC completes execution.

| interrupt levels | types of interrupts |
|---|---|
| 31 | machine check or bus error |
| 30 | power fail |
| 29 | interprocessor notification (request another processor to act; e.g., dispatch a process or update the TLB) |
| 28 | clock (used to keep track of time) |
| 27 | profile |
| 3–26 | traditional PC IRQ hardware interrupts |
| 2 | dispatch and deferred procedure call (DPC) (kernel) |
| 1 | asynchronous procedure call (APC) |
| 0 | passive |

**Figure 21.2** Windows x86 interrupt-request levels (IRLQs).

DPCs are used to postpone interrupt processing. After handling all urgent device-interrupt processing, the ISR schedules the remaining processing by queuing a DPC. The associated software interrupt runs at IRQL 2 (DPC_LEVEL), which is lower than all other hardware/I/O interrupt levels. Thus, DPCs do not block other device ISRs. In addition to deferring device-interrupt processing, the dispatcher uses DPCs to process timer expirations and to interrupt current thread execution at the end of the scheduling quantum.

Because IRQL 2 is higher than 0 (PASSIVE) and 1 (APC), execution of DPCs prevents standard threads from running on the current processor and also keeps APCs from signaling the completion of I/O. Therefore, it is important for DPC routines not to take an extended amount of time. As an alternative, the executive maintains a pool of worker threads. DPCs can queue work items to the worker threads, where they will be executed using normal thread scheduling at IRQL 0. Because the dispatcher itself runs at IRQL 2, and because paging operations require waiting on I/O (and that involves the dispatcher), DPC routines are restricted in that they cannot take page faults, call pageable system services, or take any other action that might result in an attempt to wait for a dispatcher object to be signaled. Unlike APCs, which are targeted to a thread, DPC routines make no assumptions about what process context the processor is executing, since they execute in the same context as the currently executing thread, which was interrupted.

#### 21.3.4.8 Exceptions, Interrupts, and IPIs

The kernel dispatcher also provides trap handling for exceptions and interrupts generated by hardware or software. Windows defines several architecture-independent exceptions, including:

- Integer or floating-point overflow
- Integer or floating-point divide by zero
- Illegal instruction
- Data misalignment
- Privileged instruction
- Access violation
- Paging file quota exceeded
- Debugger breakpoint

The trap handlers deal with the hardware-level exceptions (called **traps**) and call the elaborate exception-handling code performed by the kernel's exception dispatcher. The **exception dispatcher** creates an exception record containing the reason for the exception and finds an exception handler to deal with it.

When an exception occurs in kernel mode, the exception dispatcher simply calls a routine to locate the exception handler. If no handler is found, a fatal system error occurs and the user is left with the infamous “blue screen of death” that signifies system failure. In Windows 10, this is now a friendlier “sad face of sorrow” with a QR code, but the blue color remains.

Exception handling is more complex for user-mode processes, because the Windows error reporting (WER) service sets up an ALPC error port for every process, on top of the Win32 environment subsystem, which sets up an ALPC exception port for every process it creates. (For details on ports, see Section 21.3.5.4.) Furthermore, if a process is being debugged, it gets a debugger port. If a debugger port is registered, the exception handler sends the exception to the port. If the debugger port is not found or does not handle that exception, the dispatcher attempts to find an appropriate exception handler. If none exists, it contacts the default unhandled exception handler, which will notify WER of the process crash so that a crash dump can be generated and sent to Microsoft. If there is a handler, but it refuses to handle the exception, the debugger is called again to catch the error for debugging. If no debugger is running, a message is sent to the process's exception port to give the environment subsystem a chance to react to the exception. Finally, a message is sent to WER through the error port, in the case where the unhandled exception handler may not have had a chance to do so, and then the kernel simply terminates the process containing the thread that caused the exception.

WER will typically send the information back to Microsoft for further analysis, unless the user has opted out or is using a local error-reporting server. In some cases, Microsoft's automated analysis may be able to recognize the error immediately and suggest a fix or workaround.

The interrupt dispatcher in the kernel handles interrupts by calling either an interrupt service routine (ISR) supplied by a device driver or a kernel trap-handler routine. The interrupt is represented by an **interrupt object** that contains all the information needed to handle the interrupt. Using an interrupt object makes it easy to associate interrupt-service routines with an interrupt without having to access the interrupt hardware directly.

Different processor architectures have different types and numbers of interrupts. For portability, the interrupt dispatcher maps the hardware interrupts into a standard set.

The kernel uses an **interrupt-dispatch table** to bind each interrupt level to a service routine. In a multiprocessor computer, Windows keeps a separate interrupt-dispatch table (IDT) for each processor core, and each processor's IRQL can be set independently to mask out interrupts. All interrupts that occur at a level equal to or less than the IRQL of a processor are blocked until the IRQL is lowered by a kernel-level thread or by an ISR returning from interrupt processing. Windows takes advantage of this property and uses software interrupts to deliver APCs and DPCs, to perform system functions such as synchronizing threads with I/O completion, to start thread execution, and to handle timers.

### 21.3.5 Executive

The Windows executive provides a set of services that all environment subsystems use. To give you a good basic overview, we discuss the following services here: object manager, virtual memory manager, process manager, advanced local procedure call facility, I/O manager, cache manager, security reference monitor, plug-and-play and power managers, registry, and startup. Note, though, that the Windows executive includes more than two dozen services in total.

The executive is organized according to object-oriented design principles. An **object type** in Windows is a system-defined data type that has a set of attributes (data values) and a set of methods (for example, functions or operations) that help define its behavior. An **object** is an instance of an object type. The executive performs its job by using a set of objects whose attributes store the data and whose methods perform the activities.

#### 21.3.5.1 Object Manager

For managing kernel-mode entities, Windows uses a generic set of interfaces that are manipulated by user-mode programs. Windows calls these entities ***objects***, and the executive component that manipulates them is the **object manager**. Examples of objects are files, registry keys, devices, ALPC ports, drivers, mutexes, events, processes, and threads. As we saw earlier, some of these, such as mutexes and processes, are dispatcher objects, which means that threads can block in the dispatcher waiting for any of these objects to be signaled. Additionally, most of the non-dispatcher objects include an internal dispatcher object, which is signaled by the executive service controlling it. For example, file objects have an event object embedded, which is signaled when a file is modified.

User-mode and kernel-mode code can access these objects using an opaque value called a **handle**, which is returned by many APIs. Each process has a **handle table** containing entries that track the objects used by the process. There is a "system process" (see Section 21.3.5.11) that has its own handle table, which is protected from user code and is used when kernel-mode code is manipulating handles. The handle tables in Windows are represented by a tree structure, which can expand from holding 1,024 handles to holding over 16 million. In addition to using handles, kernel-mode code can also access an object by using **referenced pointer**, which it must obtain by calling a special API. When handles are used, they must eventually be closed, to avoid keeping an active reference on the object. Similarly, when kernel code uses a referenced pointer, it must use a special API to drop the reference.

A handle can be obtained by creating an object, by opening an existing object, by receiving a duplicated handle, or by inheriting a handle from a parent process. To work around the issue that developers may forget to close their handles, all of the open handles of a process are implicitly closed when it exits or is terminated. However, since kernel handles belong to the system-wide handle table, when a driver unloads, its handles are not automatically closed, and this can lead to resource leaks on the system.

Since the object manager is the only entity that generates object handles, it is the natural place to centralize calling the security reference monitor (SRM) (see Section 21.3.5.7) to check security. When an attempt is made to open an object, the object manager calls the SRM to check whether a process or thread has the right to access the object. If the access check is successful, the resulting rights (encoded as an **access mask**) are cached in the handle table. Therefore, the opaque handle both represents the object in the kernel and identifies the access that was granted to the object. This important optimization means that whenever a file is written to (which could happen hundreds of times a second), security checks are completely skipped, since the handle is already encoded as a “write” handle. Conversely, if a handle is a “read” handle, attempts to write to the file would instantly fail, without requiring a security check.

The object manager also enforces quotas, such as the maximum amount of memory a process may use, by charging a process for the memory occupied by all its referenced objects and refusing to allocate more memory when the accumulated charges exceed the process’s quota.

Because objects can be referenced through handles from user and kernel mode, and referenced through pointers from kernel mode, the object manager has to keep track of two counts for each object: the number of handles for the object and the number of references. The handle count is the number of handles that refer to the object in all of the handle tables (including the system handle table). The reference count is the sum of all handles (which count as references) plus all pointer references done by kernel-mode components. The count is incremented whenever a new pointer is needed by the kernel or a driver and decremented when the component is done with the pointer. The purpose of these reference counts is to ensure that an object is not freed while it still has a reference, but can still release some of its data (such as the name and security descriptor) when all handles are closed (since kernel-mode components don’t need this information).

The object manager maintains the Windows internal name space. In contrast to UNIX, which roots the system name space in the file system, Windows uses an abstract object manager name space that is only visible in memory or through specialized tools such as the debugger. Instead of file-system directories, the hierarchy is maintained by a special kind of object called a **directory object** that contains a hash bucket of other objects (including other directory objects). Note that some objects don’t have names (such as threads), and even for other objects, whether an object has a name is up to its creator. For example, a process would only name a mutex if it wanted other processes to find, acquire, or inquire about the state of the mutex.

Because processes and threads are created without names, they are referenced through a separate numerical identifier, such as a process ID (PID) or thread (TID). The object manager also supports symbolic links in the name space. As an example, DOS drive letters are implemented using symbolic links; `\Global??\C:` is a symbolic link to the device object `\Device\HarddiskVolumeN`, representing a mounted file-system volume in the `\Device` directory.

Each object, as mentioned earlier, is an instance of an ***object type***. The object type specifies how instances are to be allocated, how data fields are to be defined, and how the standard set of virtual functions used for all objects are to be implemented. The standard functions implement operations such as mapping names to objects, closing and deleting, and applying security checks. Functions that are specific to a particular type of object are implemented by system services designed to operate on that particular object type, not by the methods specified in the object type.

The `parse()` function is the most interesting of the standard object functions. It allows the implementation of an object to override the default naming behavior of the object manager (which is to use the virtual object directories). This ability is useful for objects that have their own internal namespace, especially when the namespace might need to be retained between boots. The

I/O manager (for file objects) and the configuration manager (for registry key objects) are the most notable users of parse functions.

Returning to our Windows naming example, device objects used to represent file-system volumes provide a parse function. This allows a name like `\Global??\C:\foo\bar.doc` to be interpreted as the file `\foo\bar.doc` on the volume represented by the device object `HarddiskVolume2`. We can illustrate how naming, parse functions, objects, and handles work together by looking at the steps to open the file in Windows:

1. An application requests that a file named `C:\foo\bar.doc` be opened.
2. The object manager finds the device object `HarddiskVolume2`, looks up the parse procedure (for example, `IopParseDevice`) from the object's type, and invokes it with the file's name relative to the root of the file system.
3. `IopParseDevice()` looks up the file system that owns the volume `HardDiskVolume2` and then calls into the file system, which looks up how to access `\foo\bar.doc` on the volume, performing its own internal parsing of the `foo` directory to find the `bar.doc` file. The file system then allocates a file object and returns it to the I/O manager's parse routine.
4. When the file system returns, the object manager allocates an entry for the file object in the handle table for the current process and returns the handle to the application.

If the file cannot successfully be opened, `IopParseDevice` returns an error indication to the application.

#### 21.3.5.2 Virtual Memory Manager

The executive component that manages the virtual address space, physical memory allocation, and paging is the **memory manager** (MM). The design of the MM assumes that the underlying hardware supports virtual-to-physical mapping, a paging mechanism, and transparent cache coherence on multiprocessor systems, as well as allowing multiple page-table entries to map to the same physical page frame. The MM in Windows uses a page-based management scheme based on the page sizes supported by hardware (4 KB, 2 MB, and 1 GB). Pages of data allocated to a process that are not in physical memory are either stored in the **paging file** on secondary storage or mapped directly to a regular file on a local or remote file system. A page can also be marked zero-fill-on-demand, which initializes the page with zeros before it is mapped, thus erasing the previous contents.

On 32-bit processors such as IA-32 and ARM, each process has a 4-GB virtual address space. By default, the upper 2 GB are mostly identical for all processes and are used by Windows in kernel mode to access the operating-system code and data structures. For 64-bit architectures such as the AMD64 architecture, Windows provides a 256-TB per-process virtual address space, divided into two 128-TB regions for user mode and kernel mode. (These restrictions are based on hardware limitations that will soon be lifted. Intel has announced that its future processors will support up to 128 PB of virtual address space, out of the 16 EB theoretically available.)

The availability of the kernel's code in each process's address space is important, and commonly found in many other operating systems as well. Generally, virtual memory is used to map the kernel code into the address space of each process. Then, when say a system call is executed or an interrupt is received, the context switch to allow the current core to run that code is lighter-weight than it would otherwise be without this mapping. Specificially, no memory-management registers need to be saved and restored, and the cache does not get invalidated. The net result is much faster movement between user and kernel code, compared to older architectures that keep kernel memory separate and not available within the process address space.

The Windows MM uses a two-step process to allocate virtual memory. The first step ***reserves*** one or more pages of virtual addresses in the process's virtual address space. The second step ***commits*** the allocation by assigning virtual memory space (physical memory or space in the paging files). Windows limits the amount of virtual memory space a process consumes by enforcing a quota on committed memory. A process de-commits memory that it is no longer using to free up virtual memory space for use by other processes. The APIs used to reserve virtual addresses and commit virtual memory take a handle on a process object as a parameter. This allows one process to control the virtual memory of another.

Windows implements shared memory by defining a **section object**. After getting a handle to a section object, a process maps the memory of the section to a range of addresses, called a **view**. A process can establish a view of the entire section or only the portion it needs. Windows allows sections to be mapped not just into the current process but into any process for which the caller has a handle.

Sections can be used in many ways. A section can be backed by secondary storage either in the system-paging file or in a regular file (a memory-mapped file). A section can be ***based***, meaning that it appears at the same virtual address for all processes attempting to access it. Sections can also represent physical memory, allowing a 32-bit process to access more physical memory than can fit in its virtual address space. Finally, the memory protection of pages in the section can be set to read only, read–write, read–write–execute, execute only, no access, or copy-on-write.

Let's look more closely at the last two of these protection settings:

- A ***no-access page*** raises an exception if accessed. The exception can be used, for example, to check whether a faulty program iterates beyond the end of an array or simply to detect that the program attempted to access virtual addresses that are not committed to memory. User- and kernel-mode stacks use no-access pages as **guard pages** to detect stack overflows. Another use is to look for heap buffer overruns. Both the user-mode memory allocator and the special kernel allocator used by the device verifier can be configured to map each allocation onto the end of a page, followed by a no-access page to detect programming errors that access beyond the end of an allocation.

- The ***copy-on-write mechanism*** enables the MM to use physical memory more efficiently. When two processes want independent copies of data from the same section object, the MM places a single shared copy into virtual memory and activates the copy-on-write property for that region of memory. If one of the processes tries to modify data in a copy-on-write page, the MM makes a private copy of the page for the process.

The virtual address translation on most modern processors uses a multilevel page table. For IA-32 (operating in Physical Address Extension, or PAE, mode) and AMD64 processors, each process has a **page directory** that contains 512 **page-directory entries** (**PDEs**), each 8 bytes in size. Each PDE points to a **PTE table** that contains 512 **page-table entries** (**PTEs**), each 8 bytes in size. Each PTE points to a 4-KB **page frame** in physical memory. For a variety of reasons, the hardware requires that the page directories or PTE tables at each level of a multilevel page table occupy a single page. Thus, the number of PDEs or PTEs that fit in a page determines how many virtual addresses are translated by that page. See Figure 21.3 for a diagram of this structure.

The structure described so far can be used to represent only 1 GB of virtual address translation. For IA-32, a second page-directory level is needed, containing only four entries, as shown in the diagram. On 64-bit processors, more entries are needed. For AMD64, the processor can fill all the remaining entries in the second page-directory level and thus obtain 512 GB of virtual address space. Therefore, to support the 256 TB that are required, the processor needs a third page-directory level (called the PML4), which also has 512 entries, each pointing to the lower-level directory. As mentioned earlier, future processors announced by Intel will support 128 PB, requiring a fourth page-directory level (PML5). Thanks to this hierarchical mechanism, the total size of all page-table pages needed to fully represent a 32-bit virtual address space for a process is

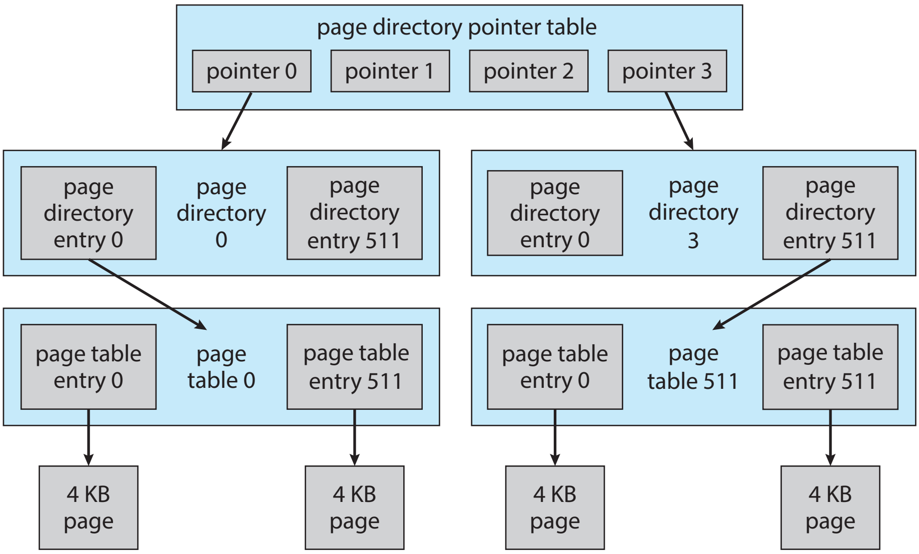

page directory pointer table
pointer 0
pointer 1
pointer 2
pointer 3
page directory entry 0
page directory 0
page directory entry 511
page directory entry 0
page directory 3
page directory entry 511
page table entry 0
page table 0
page table entry 511
page table entry 0
page table 511
page table entry 511
4 KB page
4 KB page
4 KB page
4 KB page

**Figure 21.3** Page-table layout.

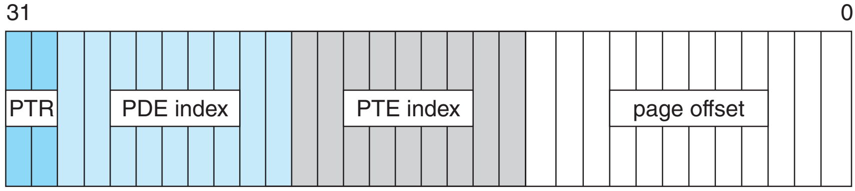


**Figure 21.4** Virtual-to-physical address translation on IA-32.

only 8 MB. Additionally, the MM allocates pages of PDEs and PTEs as needed and moves page-table pages to secondary storage when not in use, so that the actual physical memory overhead of the paging structures for each process is usually approximately 2 KB. The page-table pages are faulted back into memory when referenced.

We next consider how virtual addresses are translated into physical addresses on IA-32-compatible processors. A 2-bit value can represent the values 0, 1, 2, 3. A 9-bit value can represent values from 0 to 511; a 12-bit value, values from 0 to 4,095. Thus, a 12-bit value can select any byte within a 4-KB page of memory. A 9-bit value can represent any of the 512 PDEs or PTEs in a page directory or PTE-table page. As shown in Figure 21.4, translating a virtual address pointer to a byte address in physical memory involves breaking the 32-bit pointer into four values, starting from the most significant bits:

- Two bits are used to index into the four PDEs at the top level of the page table. The selected PDE will contain the physical page number for each of the four page-directory pages that map 1 GB of the address space.
- Nine bits are used to select another PDE, this time from a second-level page directory. This PDE will contain the physical page numbers of up to 512 PTE-table pages.
- Nine bits are used to select one of 512 PTEs from the selected PTE-table page. The selected PTE will contain the physical page number for the byte we are accessing.
- Twelve bits are used as the byte offset into the page. The physical address of the byte we are accessing is constructed by appending the lowest 12 bits of the virtual address to the end of the physical page number we found in the selected PTE.

Note that the number of bits in a physical address may be different from the number of bits in a virtual address. For example, when PAE is enabled (the only mode supported by Windows 8 and later versions), the IA-32 MMU is extended to the larger 64-bit PTE size, while the hardware supports 36-bit physical addresses, granting access to up to 64 GB of RAM, even though a single process can only map an address space up to 4 GB in size. Today, on the AMD64 architecture, server versions of Windows support very, very large physical addresses—more than we can possibly use or even buy (24 TB as of the latest release). (Of course, at one time time 4 GB seemed optimistically large for physical memory.)

To improve performance, the MM maps the page-directory and PTE-table pages into the same contiguous region of virtual addresses in every process. This self-map allows the MM to use the same pointer to access the current PDE or PTE corresponding to a particular virtual address no matter what process is running. The self-map for the IA-32 takes a contiguous 8-MB region of kernel virtual address space; the AMD64 self-map occupies 512 GB. Although the self-map occupies significant address space, it does not require any additional virtual memory pages. It also allows the page table's pages to be automatically paged in and out of physical memory.

In the creation of a self-map, one of the PDEs in the top-level page directory refers to the page-directory page itself, forming a "loop" in the page-table translations. The virtual pages are accessed if the loop is not taken, the PTE-table pages are accessed if the loop is taken once, the lowest-level page-directory pages are accessed if the loop is taken twice, and so forth.

The additional levels of page directories used for 64-bit virtual memory are translated in the same way except that the virtual address pointer is broken up into even more values. For the AMD64, Windows uses four full levels, each of which maps 512 pages, or 9 + 9 + 9 + 9 + 12 = 48 bits of virtual address.

To avoid the overhead of translating every virtual address by looking up the PDE and PTE, processors use **translation look-aside buffer** (**TLB**) hardware, which contains an associative memory cache for mapping virtual pages to PTEs. The TLB is part of the **memory-management unit** (**MMU**) within each processor. The MMU needs to "walk" (navigate the data structures of) the page table in memory only when a needed translation is missing from the TLB.

The PDEs and PTEs contain more than just physical page numbers. They also have bits reserved for operating-system use and bits that control how the hardware uses memory, such as whether hardware caching should be used for each page. In addition, the entries specify what kinds of access are allowed for both user and kernel modes.

A PDE can also be marked to say that it should function as a PTE rather than a PDE. On a IA-32, the first 11 bits of the virtual address pointer select a PDE in the first two levels of translation. If the selected PDE is marked to act as a PTE, then the remaining 21 bits of the pointer are used as the offset of the byte. This results in a 2-MB size for the page. Mixing and matching 4-KB and 2-MB page sizes within the page table is easy for the operating system and can significantly improve the performance of some programs. The improvement results from reducing how often the MMU needs to reload entries in the TLB, since one PDE mapping 2 MB replaces 512 PTEs, each mapping 4 KB. Newer AMD64 hardware even supports 1-GB pages, which operate in a similar fashion.

Managing physical memory so that 2-MB pages are available when needed is difficult, as they may continually be broken up into 4-KB pages, causing external fragmentation of memory. Also, the large pages can result in very significant internal fragmentation. Because of these problems, it is typically only Windows itself, along with large server applications, that use large pages to improve the performance of the TLB. They are better suited to do so because operating-system and server applications start running when the system boots, before memory has become fragmented.

Windows manages physical memory by associating each physical page with one of seven states: free, zeroed, modified, standby, bad, transition, or valid.

- A ***free*** page is an available page that has stale or uninitialized content.
- A ***zeroed*** page is a free page that has been zeroed out and is ready for immediate use to satisfy zero-on-demand faults.
- A ***modified*** page has been written by a process and must be sent to secondary storage before it is usable by another process.
- A ***standby*** page is a copy of information already stored on secondary storage. Standby pages may be pages that were not modified, modified pages that have already been written to secondary storage, or pages that were prefetched because they were expected to be used soon.
- A ***bad*** page is unusable because a hardware error has been detected.
- A ***transition*** page is on its way from secondary storage to a page frame allocated in physical memory.
- A ***valid*** page either is part of the working set of one or more processes and is contained within these processes' page tables, or is being used by the system directly (such as to store the nonpaged pool).

While valid pages are contained in processes' page tables, pages in other states are kept in separate lists according to state type. Additionally, to improve performance and protect against aggressive recycling of the standby pages, Windows Vista and later versions implement eight prioritized standby lists. The lists are constructed by linking the corresponding entries in the **page frame number** (**PFN**) database, which includes an entry for each physical memory page. The PFN entries also include information such as reference counts, locks, and NUMA information. Note that the PFN database represents pages of physical memory, whereas the PTEs represent pages of virtual memory.

When the valid bit in a PTE is zero, hardware ignores all the other bits, and the MM can define them for its own use. Invalid pages can have a number of states represented by bits in the PTE. Page-file pages that have never been faulted in are marked zero-on-demand. Pages mapped through section objects encode a pointer to the appropriate section object. PTEs for pages that have been written to the page file contain enough information to locate the page on secondary storage, and so forth. The structure of the page-file PTE is shown in Figure 21.5. The T, P, and V bits are all zero for this type of PTE. The PTE includes 5 bits for page protection, 32 bits for page-file offset, and 4 bits to select the paging file. There are also 20 bits reserved for additional bookkeeping.

Windows uses a per-working-set, least recently used (LRU) replacement policy to take pages from processes as appropriate. When a process is started, it is assigned a default minimum working-set size, at which point the MM starts to track the age of the pages in each working set. The working set of each process is allowed to grow until the amount of remaining physical memory starts to run low. Eventually, when the available memory runs critically low, the MM trims the working set to remove older pages.

The age of a page depends not on how long it has been in memory but on when it was last referenced. The MM makes this determination by periodically passing through the working set of each process and incrementing the age for pages that have not been marked in the PTE as referenced since the last pass. When it becomes necessary to trim the working sets, the MM uses heuristics to

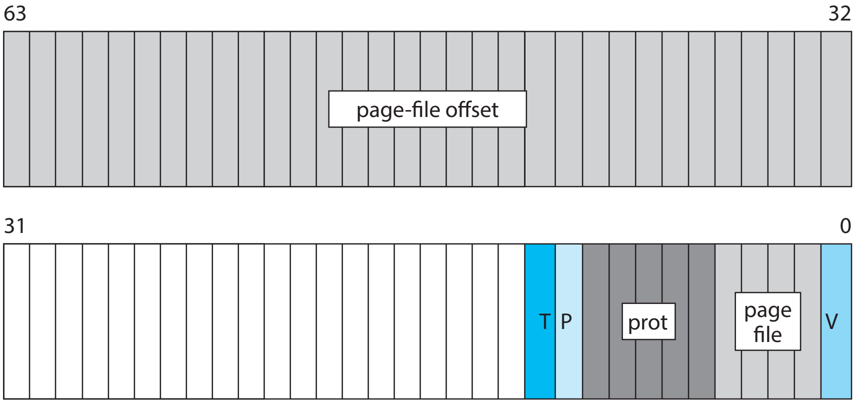


**Figure 21.5** Page-file page-table entry. The valid bit is zero.

decide how much to trim from each process and then removes the oldest pages first.

A process can have its working set trimmed even when plenty of memory is available, if it was given a ***hard limit*** on how much physical memory it could use. In Windows 7 and later versions, the MM also trims processes that are growing rapidly, even if memory is plentiful. This policy change significantly improved the responsiveness of the system for other processes.

Windows tracks working sets not only for user-mode processes but also for various kernel-mode regions, which include the file cache and the pageable kernel heap. Pageable kernel and driver code and data have their own working sets, as does each TS session. The distinct working sets allow the MM to use different policies to trim the different categories of kernel memory.

The MM does not fault in only the page immediately needed. Research shows that the memory referencing of a thread tends to have a **locality** property. That is, when a page is used, it is likely that adjacent pages will be referenced in the near future. (Think of iterating over an array or fetching sequential instructions that form the executable code for a thread.) Because of locality, when the MM faults in a page, it also faults in a few adjacent pages. This prefetching tends to reduce the total number of page faults and allows reads to be clustered to improve I/O performance.

In addition to managing committed memory, the MM manages each process's reserved memory, or virtual address space. Each process has an associated tree that describes the ranges of virtual addresses in use and what the uses are. This allows the MM to fault in page-table pages as needed. If the PTE for a faulting address is uninitialized, the MM searches for the address in the process's tree of **virtual address descriptors** (**VADs**) and uses this information to fill in the PTE and retrieve the page. In some cases, a PTE-table page may not exist; such a page must be transparently allocated and initialized by the MM. In other cases, the page may be shared as part of a section object, and the VAD will contain a pointer to that section object. The section object contains information on how to find the shared virtual page so that the PTE can be initialized to point to it directly.

Starting with Vista, the Windows MM includes a component called SuperFetch. This component combines a user-mode service with specialized kernel-mode code, including a file-system filter, to monitor all paging operations on the system. Each second, the service queries a trace of all such operations and uses a variety of agents to monitor application launches, fast user switches, standby/sleep/hibernate operations, and more as a means of understanding the system's usage patterns. With this information, it builds a statistical model, using Markov chains, of which applications the user is likely to launch when, in combination with what other applications, and what portions of these applications will be used. For example, SuperFetch can train itself to understand that the user launches Microsoft Outlook in the mornings mostly to read e-mail but composes e-mails later, after lunch. It can also understand that once Outlook is in the background, Visual Studio is likely to be launched next, and that the text editor is going to be in high demand, with the compiler demanded a little less frequently, the linker even less frequently, and the documentation code hardly ever. With this data, SuperFetch will prepopulate the standby list, making low-priority I/O reads from secondary storage at idle times to load what it thinks the user is likely to do next (or another user, if it knows a fast user switch is likely). Additionally, by using the eight prioritized standby lists that Windows offers, each such prefetched paged can be cached at a level that matches the statistical likelihood that it will be needed. Thus, unlikely-to-be-demanded pages can cheaply and quickly be evicted by an unexpected need for physical memory, while likely-to-be-demanded-soon pages can be kept in place for longer. Indeed, SuperFetch may even force the system to trim working sets of other processes before touching such cached pages.

SuperFetch's monitoring does create considerable system overhead. On mechanical (rotational) drives, which have seek times in the milliseconds, this cost is balanced by the benefit of avoiding latencies and multisecond delays in application launch times. On server systems, however, such monitoring is not beneficial, given the random multiuser workloads and the fact that throughput is more important than latency. Further, the combined latency improvements and bandwidth on systems with fast, efficient nonvolatile memory, such as SSDs, make the monitoring less beneficial for those systems as well. In such situations, SuperFetch disables itself, freeing up a few spare CPU cycles.

Windows 10 brings another large improvement to the MM by introducing a component called the compression store manager. This component creates a compressed store of pages in the working set of the **memory compression process**, which is a type of system process. When shareable pages go on the standby list and available memory is low (or certain other internal algorithm decisions are made), pages on the list will be compressed instead of evicted. This can also happen to modified pages targeted for eviction to secondary storage—both by reducing memory pressure, perhaps avoiding the write in the first place, and by causing the written pages to be compressed, thus consuming less page file space and taking less I/O to page out. On today's fast multiprocessor systems, often with built-in hardware compression algorithms, the small CPU penalty is highly preferable to the potential secondary storage I/O cost.

#### 21.3.5.3 Process Manager

The Windows process manager provides services for creating, deleting, interrogating, and managing processes, threads, and jobs. It has no knowledge about parent–child relationships or process hierarchies, although it can group processes in jobs, and the latter can have hierarchies that must then be maintained. The process manager is also not involved in the scheduling of threads, other than setting the priorities and affinities of the threads in their owner processes. Additionally, through jobs, the process manager can effect various changes in scheduling attributes (such as throttling ratios and quantum values) on threads. Thread scheduling proper, however, takes place in the kernel dispatcher.

Each process contains one or more threads. Processes themselves can be collected into larger units called **job objects**. The original use of job objects was to place limits on CPU usage, working-set size, and processor affinities that control multiple processes at once. Job objects were thus used to manage large data-center machines. In Windows XP and later versions, job objects were extended to provide security-related features, and a number of third-party applications such as Google Chrome began using jobs for this purpose. In Windows 8, a massive architectural change allowed jobs to influence scheduling through generic CPU throttling as well as per-user-session-aware fairness throttling/balancing. In Windows 10, throttling support was extended to secondary storage I/O and network I/O as well. Additionally, Windows 8 allowed job objects to nest, creating hierarchies of limits, ratios, and quotas that the system must accurately compute. Additional security and power management features were given to job objects as well.

As a result, all Windows Store applications and all UWP application processes run in jobs. The DAM, introduced earlier, implements Connected Standby support using jobs. Finally, Windows 10's support for Docker Containers, a key part of its cloud offerings, uses job objects, which it calls **silos**. Thus, jobs have gone from being an esoteric data-center resource management feature to a core mechanism of the process manager for multiple features.

Due to Windows's layered architecture and the presence of environment subsystems, process creation is quite complex. An example of process creation in the Win32 environment under Windows 10 is as follows. Note that the launching of UWP "Modern" Windows Store applications (which are called **packaged applications**, or "AppX") is significantly more complex and involves factors outside the scope of this discussion.

1. A Win32 application calls `CreateProcess()`.
2. A number of parameter conversions and behavioral conversions are done from the Win32 world to the NT world.
3. `CreateProcess()` then calls the `NtCreateUserProcess()` API in the process manager of the NT executive to actually create the process and its initial thread.
4. The process manager calls the object manager to create a process object and returns the object handle to Win32. It then calls the memory manager to initialize the address space of the new process, its handle table, and other key data structures, such as the process environment block (PEBL) (which contains internal process management data).

5. The process manager calls the object manager again to create a thread object and returns the handle to Win32. It then calls the memory manager to create the thread environment block (TEB) and the dispatcher to initialize the scheduling attributes of the thread, setting its state to `initializing`.
6. The process manager creates the initial thread startup context (which will eventually point to the `main()` routine of the application), asks the scheduler to mark the thread as `ready`, and then immediately suspends it, putting it into a `waiting` state.
7. A message is sent to the Win32 subsystem to notify it that the process is being created. The subsystem performs additional Win32-specific work to initialize the process, such as computing its shutdown level and drawing the animated hourglass or "donut" mouse cursor.
8. Back in `CreateProcess()`, inside the parent process, the `ResumeThread()` API is called to wake up the process's initial thread. Control returns to the parent.
9. Now, inside the initial thread of the new process, the user-mode link loader takes control (inside `ntdll.dll`, which is automatically mapped into all processes). It loads all the library dependencies (DLLs) of the application, creates its initial heap, sets up exception handling and application compatibility options, and eventually calls the `main()` function of the application.

The Windows APIs for manipulating virtual memory and threads and for duplicating handles take a process handle, so their subsystem and other services, when notified of process creation, can perform operations on behalf of the new process without having to execute directly in the new process's context. Windows also supports a UNIX `fork()` style of process creation. A number of features—including **process reflection**, which is used by the Windows error reporting (WER) infrastructure during process crashes, as well as the Windows subsystem for Linux's implementation of the Linux `fork()` API—depend on this capability.

The debugger support in the process manager includes the APIs to suspend and resume threads and to create threads that begin in suspended mode. There are also process-manager APIs that get and set a thread's register context and access another process's virtual memory. Threads can be created in the current process; they can also be injected into another process. The debugger makes use of thread injection to execute code within a process being debugged. Unfortunately, the ability to allocate, manipulate, and inject both memory and threads across processes is often misused by malicious programs.

While running in the executive, a thread can temporarily attach to a different process. **Thread attach** is used by kernel worker threads that need to execute in the context of the process originating a work request. For example, the MM might use thread attach when it needs access to a process's working set or page tables, and the I/O manager might use it in updating the status variable in a process for asynchronous I/O operations.

#### 21.3.5.4 Facilities for Client–Server Computing

Like many other modern operating systems, Windows uses a client–server model throughout, primarily as a layering mechanism, which allows putting common functionality into a "service" (the equivalent of a daemon in UNIX terms), as well as splitting out content-parsing code (such as a PDF reader or Web browser) from system-action-capable code (such as the Web browser's capability to save a file on secondary storage or the PDF reader's ability to print out a document). For example, on a recent Windows 10 operating system, opening the New York Times website with the Microsoft Edge browser will likely result in 12 to 16 different processes in a complex organization of "broker," "renderer/parser," "JITTer," services, and clients.

The most basic such "server" on a Windows computer is the Win32 environment subsystem, which is the server that implements the operating-system personality of the Win32 API inherited from the Windows 95/98 days. Many other services, such as user authentication, network facilities, printer spooling, Web services, network file systems, and plug-and-play, are also implemented using this model. To reduce the memory footprint, multiple services are often collected into a few processes running the `svchost.exe` program. Each service is loaded as a dynamic-link library (DLL), which implements the service by relying on user-mode thread-pool facilities to share threads and wait for messages (see Section 21.3.5.3). Unfortunately, this pooling originally resulted in poor user experience in troubleshooting and debugging runaway CPU usage and memory leaks, and it weakened the overall security of each service. Therefore, in recent versions of Windows 10, if the system has over 2 GB of RAM, each DLL service runs in its own individual `svchost.exe` process.

In Windows, the recommended paradigm for implementing client–server computing is to use RPCs to communicate requests, because of their inherent security, serialization services, and extensibility features. The Win32 API supports the Microsoft standard of the DCE-RPC protocol, called MS-RPC, as described in Section 21.6.2.7.

RPC uses multiple transports (for example, named pipes and TCP/IP) that can be used to implement RPCs between systems. When an RPC occurs only between a client and a server on the local system, ALPC can be used as the transport. Furthermore, because RPC is heavyweight and has multiple system-level dependencies (including the WINXXIII environment subsystem itself), many native Windows services, as well as the kernel, directly use ALPC, which is not available (nor suitable) for third-party programmers.

ALPC is a message-passing mechanism similar to UNIX domain sockets and Mach IPC. The server process publishes a globally visible connection-port object. When a client wants services from the server, it opens a handle to the server's connection-port object and sends a connection request to the port. If the server accepts the connection, then ALPC creates a pair of communication-port objects, providing the client's connect API with its handle to the pair, and then providing the server's accept API with the other handle to the pair.

At this point, messages can be sent across communication ports as either `datagrams`, which behave like UDP and require no reply, or `requests`, which must receive a reply. The client and server can then use either synchronous messaging, in which one side is always blocking (waiting for a request or expecting a reply), or asynchronous messaging, in which the thread-pool mechanism can be used to perform work whenever a request or reply is received, without the need for a thread to block for a message. For servers located in kernel mode, communication ports also support a callback mechanism, which allows an immediate switch to the kernel side (KT) of the user-mode thread (UT), immediately executing the server's handler routine.

When an ALPC message is sent, one of two message-passing techniques can be chosen.

1. The first technique is suitable for small to medium-sized messages (below 64 KB). In this case, the port's kernel message queue is used as intermediate storage, and the messages are copied from one process, to the kernel, to the other process. The disadvantage of this technique is the double buffering, as well as the fact that messages remain in kernel memory until the intended receiver consumes them. If the receiver is highly contended or currently unavailable, this may result in megabytes of kernel-mode memory being locked up.
2. The second technique is for larger messages. In this case, a shared-memory section object is created for the port. Messages sent through the port's message queue contain a "message attribute," called a **data view attribute**, that refers to the section object. The receiving side "exposes" this attribute, resulting in a virtual address mapping of the section object and a sharing of physical memory. This avoids the need to copy large messages or to buffer them in kernel-mode memory. The sender places data into the shared section, and the receiver sees them directly, as soon as it consumes a message.

Many other possible ways of implementing client–server communication exist, such as by using mailslots, pipes, sockets, section objects paired with events, window messages, and more. Each one has its uses, benefits, and disadvantages. RPC and ALPC remain the most fully featured, safe, secure, and feature-rich mechanisms for such communication, however, and they are the mechanisms used by the vast majority of Windows processes and services.

### 21.3.5.5 I/O Manager

The **I/O manager** is responsible for all device drivers on the system, as well as for implementing and defining the communication model that allows drivers to communicate with each other, with the kernel, and with user-mode clients and consumers. Additionally, as in UNIX-based operating systems, I/O is always targeted to a **file object**, even if the device is not a file system. The I/O manager in Windows allows device drivers to be "filtered" by other drivers, creating a **device stack** through which I/O flows and which can be used to modify, extend, or enhance the original request. Therefore, the I/O manager always keeps track of which device drivers and filter drivers are loaded.

Due to the importance of file-system drivers, the I/O manager has special support for them and implements interfaces for loading and managing file systems. It works with the MM to provide memory-mapped file I/O and controls the Windows cache manager, which handles caching for the entire I/O system. The I/O manager is fundamentally asynchronous, providing synchronous I/O by explicitly waiting for an I/O operation to complete. The I/O manager provides several models of asynchronous I/O completion, including setting of events, updating of a status variable in the calling process, delivery of APCs to initiating threads, and use of I/O completion ports, which allow a single thread to process I/O completions from many other threads. It also manages buffers for I/O requests.

Device drivers are arranged in a list for each device (called a driver or I/O stack). A driver is represented in the system as a **driver object**. Because a single driver can operate on multiple devices, the drivers are represented in the I/O stack by a **device object**, which contains a link to the driver object. Additionally, nonhardware drivers can use device objects as a way to expose different interfaces. As an example, there are TCP6, UDP6, UDP, TCP, RawIp, and RawIp6 device objects owned by the TCP/IP driver object, even though these don't represent physical devices. Similarly, each volume on secondary storage is its own device object, owned by the volume manager driver object.

Once a handle is opened to a device object, the I/O manager always creates a file object and returns a file handle instead of a device handle. It then converts the requests it receives (such as create, read, and write) into a standard form called an **I/O request packet** (**IRP**). It forwards the IRP to the first driver in the targeted I/O stack for processing. After a driver processes the IRP, it calls the I/O manager either to forward the IRP to the next driver in the stack or, if all processing is finished, to complete the operation represented by the IRP.

The I/O request may be completed in a context different from the one in which it was made. For example, if a driver is performing its part of an I/O operation and is forced to block for an extended time, it may queue the IRP to a worker thread to continue processing in the system context. In the original thread, the driver returns a status indicating that the I/O request is pending so that the thread can continue executing in parallel with the I/O operation. An IRP may also be processed in interrupt-service routines and completed in an arbitrary process context. Because some final processing may need to take place in the context that initiated the I/O, the I/O manager uses an APC to do final I/O-completion processing in the process context of the originating thread.

The I/O stack model is very flexible. As a driver stack is built, various drivers have the opportunity to insert themselves into the stack as **filter drivers**. Filter drivers can examine and potentially modify each I/O operation. Volume snapshotting (**shadow copies**) and disk encryption (**BitLocker**) are two built-in examples of functionality implemented using filter drivers that execute above the volume manager driver in the stack. File-system filter drivers execute above the file system and have been used to implement functionalities such as hierarchical storage management, single instancing of files for remote boot, and dynamic format conversion. Third parties also use file-system filter drivers to implement anti-malware tools. Due to the large number of file-system filters, Windows Server 2003 and later versions now include a **filter manager** component, which acts as the sole file-system filter and which loads **minifilters** ordered by specific **altitudes** (relative priorities). This model allows filters to transparently cache data and repeated queries without having to know about each other's requests. It also provides stricter load ordering.

Device drivers for Windows are written to the Windows Driver Model (WDM) specification. This model lays out all the requirements for device drivers, including how to layer filter drivers, share common code for handling power and plug-and-play requests, build correct cancellation logic, and so forth.

Because of the richness of the WDM, writing a full WDM device driver for each new hardware device can involve a great deal of work. In some cases, the port/miniport model makes it unnecessary to do this for certain hardware devices. Within a range of devices that require similar processing, such as audio drivers, storage controllers, or Ethernet controllers, each instance of a device shares a common driver for that class, called a **port driver**. The port driver implements the standard operations for the class and then calls device-specific routines in the device's **miniport driver** to implement device-specific functionality. The physical-link layer of the network stack is implemented in this way, with the `ndis.sys` port driver implementing much of the generic network processing functionality and calling out to the network miniport drivers for specific hardware commands related to sending and receiving network frames (such as Ethernet).

Similarly, the WDM includes a class/miniclass model. Here, a certain class of devices can be implemented in a generic way by a single class driver, with callouts to a miniclass for specific hardware functionality. For example, the Windows disk driver is a class driver, as are drivers for CD/DVDs and tape drives. The keyboard and mouse driver are class drivers as well. These types of devices don't need a miniclass, but the battery class driver, for example, does require a miniclass for each of the various external uninterruptible power supplies (UPSs) sold by vendors.

Even with the port/miniport and class/miniclass model, significant kernel-facing code must be written. And this model is not useful for custom hardware or for logical (nonhardware) drivers. Starting with Windows 2000 Service Pack 4, kernel-mode drivers can be written using the **Kernel-Mode Driver Framework** (**KMDF**), which provides a simplified programming model for drivers on top of WDM. Another option is the **User-Mode Driver Framework** (**UMDF**), which allows drivers to be written in user mode through a **reflector** driver in the kernel that forwards the requests through the kernel's I/O stack. These two frameworks make up the **Windows Driver Foundation** model, which has reached Version 2.1 in Windows 10 and contains a fully compatible API between KMDF and UMDF. It has been fully open-sourced on GitHub.

Because many drivers do not need to operate in kernel mode, and it is easier to develop and deploy drivers in user mode, UMDF is strongly recommended for new drivers. It also makes the system more reliable, because a failure in a user-mode driver does not cause a kernel (system) crash.

#### 21.3.5.6 Cache Manager

In many operating systems, caching is done by the block device system, usually at the physical/block level. Instead, Windows provides a centralized caching facility that operates at the logical/virtual file level. The **cache manager** works closely with the MM to provide cache services for all components under the control of the I/O manager. This means that the cache can operate on anything from remote files on a network share to logical files on a custom file system. The size of the cache changes dynamically according to how much free memory is available in the system; it can grow as large as 2 TB on a 64-bit system.

The cache manager maintains a private working set rather than sharing the system process's working set, which allows trimming to page out cached files more effectively. To build the cache, the cache manager memory-maps files into kernel memory and then uses special interfaces to the MM to fault pages into or trim them from this private working set, which lets it take advantage of additional caching facilities provided by the memory manager.

The cache is divided into blocks of 256 KB. Each cache block can hold a view (that is, a memory-mapped region) of a file. Each cache block is described by a **virtual address control block** (**VACB**) that stores the virtual address and file offset for the view, as well as the number of processes using the view. The VACBs reside in arrays maintained by the cache manager, and there are arrays for critical as well as low-priority cached data to improve performance in situations of memory pressure.

When the I/O manager receives a file's user-level read request, the I/O manager sends an IRP to the I/O stack for the volume on which the file resides. For files that are marked as cacheable, the file system calls the cache manager to look up the requested data in its cached file views. The cache manager calculates which entry of that file's VACB index array corresponds to the byte offset of the request. The entry either points to the view in the cache or is invalid. If it is invalid, the cache manager allocates a cache block (and the corresponding entry in the VACB array) and maps the view into the cache block. The cache manager then attempts to copy data from the mapped file to the caller's buffer. If the copy succeeds, the operation is completed.

If the copy fails, it does so because of a page fault, which causes the MM to send a noncached read request to the I/O manager. The I/O manager sends another request down the driver stack, this time requesting a paging operation, which bypasses the cache manager and reads the data from the file directly into the page allocated for the cache manager. Upon completion, the VACB is set to point at the page. The data, now in the cache, are copied to the caller's buffer, and the original I/O request is completed. Figure 21.6 shows an overview of these operations.

When possible, for synchronous operations on cached files, I/O is handled by the **fast I/O mechanism**. This mechanism parallels the normal IRP-based I/O

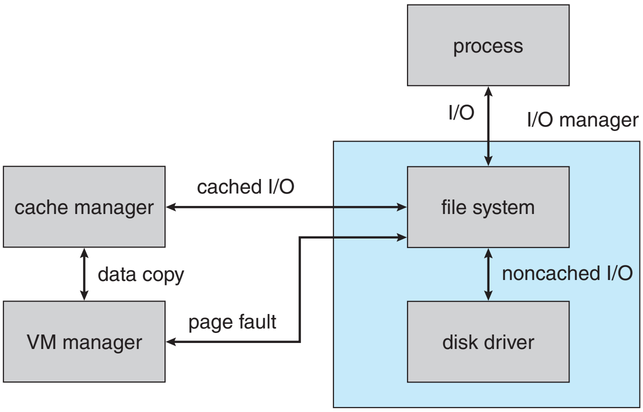


**Figure 21.6** File I/O.

but calls into the driver stack directly rather than passing down an IRP, which saves memory and time. Because no IRP is involved, the operation should not block for an extended period of time and cannot be queued to a worker thread. Therefore, when the operation reaches the file system and calls the cache manager, the operation fails if the information is not already in the cache. The I/O manager then attempts the operation using the normal IRP path.

A kernel-level read operation is similar, except that the data can be accessed directly from the cache rather than being copied to a buffer in user space. To use file-system metadata (data structures that describe the file system), the kernel uses the cache manager's mapping interface to read the metadata. To modify the metadata, the file system uses the cache manager's pinning interface. **Pinning** a page locks the page into a physical-memory page frame so that the MM manager cannot move the page or page it out. After updating the metadata, the file system asks the cache manager to unpin the page. A modified page is marked dirty, and so the MM flushes the page to secondary storage.

To improve performance, the cache manager keeps a small history of read requests and from this history attempts to predict future requests. If the cache manager finds a pattern in the previous three requests, such as sequential access forward or backward, it prefetches data into the cache before the next request is submitted by the application. In this way, the application may find its data already cached and not need to wait for secondary storage I/O.

The cache manager is also responsible for telling the MM to flush the contents of the cache. The cache manager's default behavior is write-back caching: it accumulates writes for 4 to 5 seconds and then wakes up the cache-writer thread. When write-through caching is needed, a process can set a flag when opening the file, or can call an explicit cache-flush function.

A fast-writing process could potentially fill all the free cache pages before the cache-writer thread had a chance to wake up and flush the pages to secondary storage. The cache writer prevents a process from flooding the system in the following way. When the amount of free cache memory becomes low, the cache manager temporarily blocks processes attempting to write data and wakes the cache-writer thread to flush pages to secondary storage. If the fast-writing process is actually a network redirector for a network file system, blocking it for too long could cause network transfers to time out and be retransmitted. This retransmission would waste network bandwidth. To prevent such waste, network redirectors can instruct the cache manager to limit the backlog of writes in the cache.

Because a network file system needs to move data between secondary storage and the network interface, the cache manager also provides a DMA interface to move the data directly. Moving data directly avoids the need to copy data through an intermediate buffer.

#### 21.3.5.7 Security Reference Monitor

Centralizing management of system entities in the object manager enables Windows to use a uniform mechanism to perform run-time access validation and audit checks for every user-accessible entity in the system. Additionally, even entities not managed by the object manager may have access to the API routines for performing security checks. Whenever a thread opens a handle to a protected data structure (such as an object), the **security reference monitor**

(**SRM**) checks the effective security token and the object's security descriptor, which contains two access-control lists—the discretionary access control list (DACL) and the system access control list (SACL)—to see whether the process has the necessary access rights. The effective security token is typically the token of the thread's process, but it can also be the token of the thread itself, as described below.

Each process has an associated **security token**. When the login process (`lsass.exe`) authenticates a user, the security token is attached to the user's first process (`userinit.exe`) and copied for each of its child processes. The token contains the **security identity** (**SID**) of the user, the SIDs of the groups the user belongs to, the privileges the user has, the integrity level of the process, the attributes and claims associated with the user, and any relevant capabilities. By default, threads don't have their own explicit tokens, causing them to share the common token of the process. However, using a mechanism called **impersonation**, a thread running in a process with a security token belonging to one user can set a thread-specific token belonging to another user to impersonate that user. At this point, the effective token becomes the token of the thread, and all operations, quotas, and limitations are subject to that user's token. The thread can later choose to "revert" to its old identity by removing the thread-specific token, so that the effective token is once again that of the process.

This impersonation facility is fundamental to the client–server model, where services must act on behalf of a variety of clients with different security IDs. The right to impersonate a user is most often delivered as part of a connection from a client process to a server process. Impersonation allows the server to access system services as if it were the client in order to access or create objects and files on behalf of the client. The server process must be trustworthy and must be carefully written to be robust against attacks. Otherwise, one client could take over a server process and then impersonate any user who made a subsequent client request. Windows provides APIs to support impersonation at the ALPC (and thus RPC and DCOM) layer, the named pipe layer, and the Winsock layer.

The SRM is also responsible for manipulating the privileges in security tokens. Special privileges are required for users to change the system time, load a driver, or change firmware environment variables. Additionally, certain users can have powerful privileges that override default access control rules. These include users who must perform backup or restore operations on file systems (allowing them to bypass read/write restrictions), debug processes (allowing them to bypass security features), and so forth.

The integrity level of the code executing in a process is also represented by a token. Integrity levels are a type of mandatory labeling mechanism, as mentioned earlier. By default, a process cannot modify an object with an integrity level higher than that of the code executing in the process, whatever other permissions have been granted. In addition, it cannot read from another process object at a higher integrity level. Objects can also protect themselves from read access by manually changing the mandatory policy associated with their security descriptor. Inside an object (such as a file or a process), the integrity level is stored in the SACL, which distinguishes it from typical discretionary user and group permissions, stored in the DACL.

Integrity levels were introduced to make it harder for code to take over a system by attacking external-content-parsing software, like a browser or

PDF reader, because such software is expected to run at a low integrity level. For example, Microsoft Edge runs at “low integrity,” as do Adobe Reader and Google Chrome. A regular application, such as Microsoft Word, runs at “medium integrity.” Finally, you can expect an application run by an administrator or a setup program to run at “high integrity.”

Creating applications to run at lower integrity levels places a burden on the developers to implement this security feature, because they must create a client–server model to support a broker and parser or renderer, as mentioned earlier. In order to streamline this security model, Windows 8 introduced the **Application Container**, often just called “AppContainer,” which is a special extension of the token object. When running under an AppContainer, an application automatically has its process token adjusted in the following ways:

1. The token’s integrity level is set to low. This means that the application cannot write to or modify most objects (files, keys, processes) on the system, nor can it read from any other process on the system.
2. All groups and the user SID are disabled (ignored) in the token. Let’s say that the application was launched by user Anne, who belongs to the World group. Any files accessible to Anne or World will be inaccessible to this application.
3. All privileges except a handful are removed from the token. This prevents powerful system calls or system-wide operations from being permitted.
4. A special AppContainer SID is added to the token, which corresponds to the SHA-256 hash of the application’s package identifier. This is the only valid security identifier in the token, so any object wishing to be directly accessible to this application needs to explicitly give the AppContainer SID read or write access.
5. A set of capability SIDs are added to the token, based on the application’s manifest file. When the application is first installed, these capabilities are shown to the user, who must agree to them before the application is deployed.

We can see that the AppContainer mechanism changes the security model from a discretionary system where access to protected resources is defined by users and groups to a mandatory system where each application has its own unique security identity and access occurs on a per-application basis. This separation of privileges and permissions is a great leap forward in security, but it places a potential burden on resource access. Capabilities and brokers help to alleviate this burden.

Capabilities are used by system brokers implemented by Windows to perform various actions on behalf of packaged applications. For example, assume that Harold’s packaged application has no access to Harold’s file system, since the Harold SID is disabled. In this situation, a broker might check for the `Play User Media` capability and allow the music player process to read any MP3 files located in Harold’s `My Music` directory. Thus, Harold will not be forced to mark all of his files with the AppContainer SID of his favorite media player application, as long as the application has the `Play User Media` capability and Harold agreed to it when he downloaded the application.

A final responsibility of the SRM is logging security audit events. The ISO standard **Common Criteria** (the international successor to the Orange Book standard developed by the United States Department of Defense) requires that a secure system have the ability to detect and log all attempts to access system resources so that it can more easily trace attempts at unauthorized access. Because the SRM is responsible for making access checks, it generates most of the audit records, which are then written by `lsass.exe` into the security-event log.

#### 21.3.5.8 Plug-and-Play Manager

The operating system uses the **plug-and-play** (**PnP**) **manager** to recognize and adapt to changes in hardware configuration. PnP devices use standard protocols to identify themselves to the system. The PnP manager automatically recognizes installed devices and detects changes in devices as the system operates. The manager also keeps track of hardware resources used by a device, as well as potential resources that could be used, and takes care of loading the appropriate drivers. This management of hardware resources—primarily interrupts, DMA channels, and I/O memory ranges—has the goal of determining a hardware configuration in which all devices are able to operate successfully. The PnP manager and the Windows Driver Model see drivers as either `bus drivers`, which detect and enumerate the devices on a bus (such as PCI or USB), or `function drivers`, which implement the functionality of a particular device on the bus.

The PnP manager handles dynamic reconfiguration as follows. First, it gets a list of devices from each bus driver. It loads the drivers and sends an `add-device` request to the appropriate driver for each device. Working in tandem with special **resource arbiters** owned by the various bus drivers, the PnP manager then figures out the optimal resource assignments and sends a `start-device` request to each driver specifying the resource assignments for the related devices. If a device needs to be reconfigured, the PnP manager sends a `query-stop` request, which asks the driver whether the device can be temporarily disabled. If the driver can disable the device, then all pending operations are completed, and new operations are prevented from starting. Finally, the PnP manager sends a `stop` request and can then reconfigure the device with a new `start-device` request.

The PnP manager also supports other requests. For example, `query-remove`, which operates similarly to `query-stop`, is employed when a user is getting ready to eject a removable device, such as a USB storage device. The `surprise-remove` request is used when a device fails or, more often, when a user removes a device without telling the system to stop it first. Finally, the `remove` request tells the driver to stop using a device permanently.

Many programs in the system are interested in the addition or removal of devices, so the PnP manager supports notifications. Such a notification, for example, gives the file manager the information it needs to update its list of secondary storage volumes when a new storage device is attached or removed.

Installing devices can also result in starting new services on the system. Previously, such services frequently set themselves up to run whenever the system booted and continued to run even if the associated device was never plugged into the system, because they had to be running in order to receive the

PnP notification. Windows 7 introduced a **service-trigger** mechanism in the **service control manager** (**SCM**) (`services.exe`), which manages the system services. With this mechanism, services can register themselves to start only when SCM receives a notification from the PnP manager that the device of interest has been added to the system.

### 21.3.5.9 Power Manager

Windows works with the hardware to implement sophisticated strategies for energy efficiency, as described in Section 21.2.8. The policies that drive these strategies are implemented by the **power manager**. The power manager detects current system conditions, such as the load on CPUs or I/O devices, and improves energy efficiency by reducing the performance and responsiveness of the system when need is low. The power manager can also put the entire system into a very efficient ***sleep*** mode and can even write all the contents of memory to secondary storage and turn off the power to allow the system to go into ***hibernation***.

The primary advantage of sleep is that the system can enter that state fairly quickly, perhaps just a few seconds after the lid closes on a laptop. The return from sleep is also fairly quick. The power is turned down to a low level on the CPUs and I/O devices, but the memory continues to be powered enough that its contents are not lost. As noted earlier, however, on mobile devices, these few seconds still add up to an unreasonable user experience, so the power manager works with the Desktop Activity Moderator to kick off the Connected Standby state as soon as the screen is turned off. Connected Standby virtually freezes the computer but does not really put the computer to sleep.

Hibernation takes considerably longer to enter than sleep because the entire contents of memory must be transferred to secondary storage before the system is turned off. However, the fact that the system is, in fact, turned off is a significant advantage. If there is a loss of power to the system, as when the battery is swapped on a laptop or a desktop system is unplugged, the saved system data will not be lost. Unlike shutdown, hibernation saves the currently running system so a user can resume where she left off. Furthermore, because hibernation does not require power, a system can remain in hibernation indefinitely. Therefore, this feature is extremely useful on desktops and server systems, and it is also used on laptops when the battery hits a critical level (because putting the system to sleep when the battery is low might result in the loss of all data if the battery runs out of power while in the sleep state).

In Windows 7, the power manager also includes a processor power manager (PPM), which specifically implements strategies such as core parking, CPU throttling and boosting, and more. In addition, Windows 8 introduced the **power framework** (**PoFX**), which works with function drivers to implement specific functional power states. This means that devices can expose their internal power management (clock speeds, current/power draws, and so forth) to the system, which can then use the information for fine-grained control of the devices. Thus, for example, instead of simply turning a device on or off, the system can turn specific components on or off.

Like the PnP manager, the power manager provides notifications to the rest of the system about changes in the power state. Some applications want to know when the system is about to be shut down so they can start saving their states to secondary storage, and, as mentioned earlier, the DAM needs to know when the screen is turned off and on again.

#### 21.3.5.10 Registry

Windows keeps much of its configuration information in internal repositories of data, called **hives**, that are managed by the Windows configuration manager, commonly known as the **registry**. The configuration manager is implemented as a component of the executive.

There are separate hives for system information, each user's preferences, software information, security, and boot options. Additionally, as part of the new application and security model introduced by AppContainers and UWPModern/Metro packaged applications in Windows 8, each such application has its own separate hive, called an application hive.

The registry represents the configuration state in each hive as a hierarchical namespace of keys (directories), each of which can contain a set of arbitrarily sized values. In the Win32 API, these values have a specific "type," such as UNICODE string, 32-bit integer, or untyped binary data, but the registry itself treats all values the same, leaving it up to the higher API layers to infer a structure based on type and size. Therefore, for example, nothing prevents a "32-bit integer" from being a 999-byte UNICODE string.

In theory, new keys and values are created and initialized as new software is installed, and then they are modified to reflect changes in the configuration of that software. In practice, the registry is often used as a general-purpose database, as an interprocess-communication mechanism, and for many other such inventive purposes.

Restarting applications, or even the system, every time a configuration change was made would be a nuisance. Instead, programs rely on various kinds of notifications, such as those provided by the PnP and power managers, to learn about changes in the system configuration. The registry also supplies notifications; threads can register to be notified when changes are made to some part of the registry. The threads can thus detect and adapt to configuration changes recorded in the registry. Furthermore, registry keys are objects managed by the object manager, and they expose an event object to the dispatcher. This allows threads to put themselves in a `waiting` state associated with the event, which the configuration manager will signal if the key (or any of its values) is ever modified.

Whenever significant changes are made to the system, such as when updates to the operating system or drivers are installed, there is a danger that the configuration data may be corrupted (for example, if a working driver is replaced by a nonworking driver or an application fails to install correctly and leaves partial information in the registry). Windows creates a **system restore point** before making such changes. The restore point contains a copy of the hives before the change and can be used to return to this version of the hives in order to get a corrupted system working again.

To improve the stability of the registry configuration, the registry also implements a variety of "self-healing" algorithms, which can detect and fix certain cases of registry corruption. Additionally, the registry internally uses a two-phase commit transactional algorithm, which prevents corruption to individual keys or values as they are being updated. While these mechanisms guarantee the integrity of small portions of the registry or individual keys and values, they have not supplanted the system restore facility for recovering from damage to the registry configuration caused by a failure during a software installation.

#### 21.3.5.11 Booting

The booting of a Windows PC begins when the hardware powers on and firmware begins executing from ROM. In older machines, this firmware was known as the BIOS, but more modern systems use UEFI (the Unified Extensible Firmware Interface), which is faster, is more modern, and makes better use of the facilities in contemporary processors. Additionally, UEFI includes a feature called **Secure Boot** that provides integrity checks through digital signature verification of all firmware and boot-time components. This digital signature check guarantees that only Microsoft's boot-time components and the vendor's firmware are present at boot time, preventing any early third-party code from loading.

The firmware runs **power-on self-test** (**POST**) diagnostics, identifies many of the devices attached to the system and initializes them to a clean power-up state, and then builds the description used by ACPI. Next, the firmware finds the system boot device, loads the Windows boot manager program (`bootmgfw.efi` on UEFI systems), and begins executing it.

In a machine that has been hibernating, the `winresume.efi` program is loaded next. It restores the running system from secondary storage, and the system continues execution at the point it had reached right before hibernating. In a machine that has been shut down, `bootmgfw.efi` performs further initialization of the system and then loads `winload.efi`. This program loads `hal.dll`, the kernel (`ntoskrnl.exe`) and its dependencies, and any drivers needed in booting, and the system hive. `winload` then transfers execution to the kernel.

The procedure is somewhat different on Windows 10 systems where Virtual Secure Mode is enabled (and the hypervisor is turned on). Here, `winload.efi` will instead load `hvloader.exe` or `hvloader.dll`, which initializes the hypervisor first. On Intel systems, this is `hvix64.exe`, while AMD systems use `hvax64.exe`. The hypervisor then sets up VTL 1 (the Secure World) and VTL 0 (the Normal World) and returns to `winload.efi`, which now loads the secure kernel (`securekernel.exe`) and its dependencies. Then the secure kernel's entry point is called, which initializes VTL 1, after which it returns back to the loader at VTL 0, which resumes with the steps described above.

As the kernel initializes itself, it creates several processes. The **idle process** serves as the container of all idle threads, so that system-wide CPU idle time can easily be computed. The **system process** contains all of the internal kernel worker threads and other system threads created by drivers for polling, housekeeping, and other background work. The memory compression process, new to Windows 10, has a working set composed of compressed standby pages used by the store manager to alleviate system pressure and optimize paging. Finally, if VSM is enabled, the **secure system process** represents the fact that the secure kernel is loaded.

The first user-mode process, which is also created by the kernel, is **session manager subsystem** (**SMSS**), which is similar to the `init` (initialization) process in UNIX. SMSS performs further initialization of the system, including establishing the paging files and creating the initial user sessions. Each session represents a logged-on user, except for ***session 0***, which is used to run system-wide background processes, such as `lsass` and `services`. Each session is given its own instance of an SMSS process, which exits once the session is created. In each of these sessions, this ephemeral SMSS loads the Win32 environment subsystem (`csrss.exe`) and its driver (`win32k.sys`). Then, in each session other than 0, SMSS runs the `winlogon` process, which launches `logonui`. This process captures user credentials in order for `lsass` to log in a user, then launch the `userinit` and `explorer` process, which implements the Windows shell (start menu, desktop, tray icons, notification center, and so on). The following list itemizes some of these aspects of booting:

- SMSS completes system initialization and then starts up one SMSS for session 0 and one SMSS for the first login session (1).
- `wininit` runs in session 0 to initialize user mode and start `lsass` and `services`.
- `lsass`, the security subsystem, implements facilities such as authentication of users. If user credentials are protected by VSMthrough Credential Guard, then `lsaiso` and `bioiso` are also started as VTL 1 Trustlets by `lsass`.
- `services` contains the service control manager, or SCM, which supervises all background activities in the system, including user-mode services. A number of services will have registered to start when the system boots. Others will be started only on demand or when triggered by an event such as the arrival of a device.
- `csrss` is the Win32 environment subsystem process. It is started in every session—mainly because it handles mouse and keyboard input, which needs to be separated per user.
- `winlogon` is run in each Windows session other than session 0 to log on a user by launching `logonui`, which presents the logon user interface.

Starting with Windows XP, the system optimizes the boot process by prefetching pages from files on secondary storage based on previous boots of the system. Disk access patterns at boot are also used to lay out system files on disk to reduce the number of I/O operations required. Windows 7 reduced the processes necessary to start the system by allowing services to start only when needed, rather than at system start-up. Windows 8 further reduced boot time by parallelizing all driver loads through a pool of worker threads in the PnP subsystem and by supporting UEFI to make boot-time transition more efficient. All of these approaches contributed to a dramatic reduction in system boot time, but eventually little further improvement was possible.

To address boot-time concerns, especially on mobile systems, where RAM and cores are limited, Windows 8 also introduced **Hybrid Boot**. This feature combines hibernation with a simple logoff of the current user. When the user shuts down the system, and all other applications and sessions have exited, the system is returned to the `logonui` prompt and then is hibernated. When the system is turned on again, it resumes very quickly to the logon screen, which gives drivers a chance to reinitialize devices and gives the appearance of a full boot while work is still occurring.

## 21.4 Terminal Services and Fast User Switching

Windows supports a GUI-based console that interfaces with the user via keyboard, mouse, and display. Most systems also support audio and video. For example, audio input is used by Cortana, Windows's voice-recognition and virtual assistant software, which is powered by machine learning. Cortana makes the system more convenient and can also increase its accessibility for users with motor disabilities. Windows 7 added support for **multi-touch hardware**, allowing users to input data by touching the screen with one or more fingers. Video-input capability is used both for accessibility and for security: Windows Hello is a security feature in which advanced 3D heat-sensing, face-mapping cameras and sensors can be used to uniquely identify the user without requiring traditional credentials. In newer versions of Windows 10, eye-motion sensing hardware—in which mouse input is replaced by information on the position and gaze of the eyeballs—can be used for accessibility. Other future input experiences will likely evolve from Microsoft's **HoloLens** augmented-reality product.

The PC was, of course, envisioned as a ***personal computer***—an inherently single-user machine. For some time, however, Windows has supported the sharing of a PC among multiple users. Each user who is logged on using the GUI has a **session** created to represent the GUI environment he will be using and to contain all the processes necessary to run his applications. Windows allows multiple sessions to exist at the same time on a single machine. However, client versions of Windows support only a single console, consisting of all the monitors, keyboards, and mice connected to the PC. Only one session can be connected to the console at a time. From the logon screen displayed on the console, users can create new sessions or attach to an existing session. This allows multiple users to share a single PC without having to log off and on between users. Microsoft calls this use of sessions ***fast user switching.*** macOS has a similar feature.

A user on one PC can also create a new session or connect to an existing session on another computer, which becomes a **remote desktop**. The terminal services feature (TS) makes the connection through a protocol called Remote Desktop Protocol (RDP). Users often employ this feature to connect to a session on a work PC from a home PC. Remote desktops can also be used for remote troubleshooting scenarios: a remote user can be invited to share a session with the user logged on to the session on the console. The remote user can watch the user's actions and can even be given control of the desktop to help resolve computing problems. This latter use of terminal services uses the "mirroring" feature, where the alternative user is sharing the same session instead of creating a separate one.

Many corporations use corporate systems maintained in data centers to run all user sessions that access corporate resources, rather than allowing users to access those resources from their PCs, by exclusively dedicating these machines as terminal servers. Each server computer may handle hundreds of remote-desktop sessions. This is a form of **thin-client computing**, in which individual computers rely on a server for many functions. Relying on data-center terminal servers improves the reliability, manageability, and security of corporate computing resources.

## 21.5 File System

The native file system in Windows is NTFS. It is used for all local volumes. However, associated USB thumb drives, flash memory on cameras, and external storage devices may be formatted with the 32-bit FAT file system for portability. FAT is a much older file-system format that is understood by many systems besides Windows, such as the software running on cameras. A disadvantage is that the FAT file system does not restrict file access to authorized users. The only solution for securing data with FAT is to run an application to encrypt the data before storing it on the file system.

In contrast, NTFS uses ACLs to control access to individual files and supports implicit encryption of individual files or entire volumes (using Windows BitLocker feature). NTFS implements many other features as well, including data recovery, fault tolerance, very large files and file systems, multiple data streams, UNICODE names, sparse files, journaling, volume shadow copies, and file compression.

### 21.5.1 NTFS Internal Layout

The fundamental entity in NTFS is the volume. A volume is created by the Windows logical disk management utility and is based on a logical disk partition. A volume may occupy a portion of a device or an entire device, or may span several devices. The volume manager can protect the contents of the volume with various levels of RAID.

NTFS does not deal with individual sectors of a storage device but instead uses **clusters** as the units of storage allocation. The cluster size, which is a power of 2, is configured when an NTFS file system is formatted. The default cluster size is based on the volume size—4 KB for volumes larger than 2 GB. Given the size of today's storage devices, it may make sense to use cluster sizes larger than the Windows defaults to achieve better performance, although these performance gains will come at the expense of more internal fragmentation.

NTFS uses **logical cluster numbers** (**LCNs**) as storage addresses. It assigns them by numbering clusters from the beginning of the device to the end. Using this scheme, the system can calculate a physical storage offset (in bytes) by multiplying the LCN by the cluster size.

A file in NTFS is not a simple byte stream as it is in UNIX; rather, it is a structured object consisting of typed **attributes**. Each attribute of a file is an independent byte stream that can be created, deleted, read, and written. Some attribute types are standard for all files, including the file name (or names, if the file has aliases, such as an MS-DOS short name), the creation time, and the security descriptor that specifies the access control list. User data are stored in ***data attributes***.

Most traditional data files have an ***unnamed*** data attribute that contains all the file's data. However, additional data streams can be created with explicit names. The IProp interfaces of the Component Object Model (discussed later in this chapter) use a named data stream to store properties on ordinary files, including thumbnails of images. In general, attributes can be added as necessary and are accessed using a ***file-name:attribute*** syntax. NTFS returns only the size of the unnamed attribute in response to file-query operations, such as when running the `dir` command.

Every file in NTFS is described by one or more records in an array stored in a special file called the master file table (MFT). The size of a record is determined when the file system is created; it ranges from 1 to 4 KB. Small attributes are stored in the MFT record itself and are called ***resident attributes***. Large attributes, such as the unnamed bulk data, are called ***nonresident attributes*** and are stored in one or more contiguous extents on the device. A pointer to each extent is stored in the MFT record. For a small file, even the data attribute may fit inside the MFT record. If a file has many attributes—or if it is highly fragmented, so that many pointers are needed to point to all the fragments—one record in the MFT might not be large enough. In this case, the file is described by a record called the **base file record**, which contains pointers to overflow records that hold the additional pointers and attributes.

Each file in an NTFS volume has a unique ID called a **file reference**. The file reference is a 64-bit quantity that consists of a 48-bit file number and a 16-bit sequence number. The file number is the record number (that is, the array slot) in the MFT that describes the file. The sequence number is incremented every time an MFT entry is reused. The sequence number enables NTFS to perform internal consistency checks, such as catching a stale reference to a deleted file after the MFT entry has been reused for a new file.

#### 21.5.1.1 NTFS B+ Tree

As in UNIX, the NTFS namespace is organized as a hierarchy of directories. Each directory uses a data structure called a **B+ tree** to store an index of the file names in that directory. In a B+ tree, the length of every path from the root of the tree to a leaf is the same, and the cost of reorganizing the tree is eliminated. The **index root** of a directory contains the top level of the B+ tree. For a large directory, this top level contains pointers to disk extents that hold the remainder of the tree. Each entry in the directory contains the name and file reference of the file, as well as a copy of the update timestamp and file size taken from the file's resident attributes in the MFT. Copies of this information are stored in the directory so that a directory listing can be efficiently generated. Because all the file names, sizes, and update times are available from the directory itself, there is no need to gather these attributes from the MFT entries for each of the files.

#### 21.5.1.2 NTFS Metadata

The NTFS volume's metadata are all stored in files. The first file is the MFT. The second file, which is used during recovery if the MFT is damaged, contains a copy of the first 16 entries of the MFT. The next few files are also special in purpose. They include the following files:

- The **log file** records all metadata updates to the file system.
- The **volume file** contains the name of the volume, the version of NTFS that formatted the volume, and a bit that tells whether the volume may have

been corrupted and needs to be checked for consistency using the `chkdsk` program.

- The **attribute-definition table** indicates which attribute types are used in the volume and what operations can be performed on each of them.
- The **root directory** is the top-level directory in the file-system hierarchy.
- The **bitmap file** indicates which clusters on a volume are allocated to files and which are free.
- The **boot file** contains the startup code for Windows and must be located at a particular secondary storage device address so that it can be found easily by a simple ROM bootstrap loader. The boot file also contains the physical address of the MFT.
- The **bad-cluster file** keeps track of any bad areas on the volume; NTFS uses this record for error recovery.

Keeping all the NTFS metadata in actual files has a useful property. As discussed in Section 21.3.5.6, the cache manager caches file data. Since all the NTFS metadata reside in files, these data can be cached using the same mechanisms used for ordinary data.

### 21.5.2 Recovery

In many simple file systems, a power failure at the wrong time can damage the file-system data structures so severely that the entire volume is scrambled. Many UNIX file systems, including UFS but not ZFS, store redundant metadata on the storage device, and they recover from crashes by using the `fsck` program to check all the file-system data structures and restore them forcibly to a consistent state. Restoring them often involves deleting damaged files and freeing data clusters that had been written with user data but not properly recorded in the file system's metadata structures. This checking can be a slow process and can result in the loss of significant amounts of data.

NTFS takes a different approach to file-system robustness. In NTFS, all file-system data-structure updates are performed inside transactions. Before a data structure is altered, the transaction writes a log record that contains redo and undo information. After the data structure has been changed, the transaction writes a commit record to the log to signify that the transaction succeeded.

After a crash, the system can restore the file-system data structures to a consistent state by processing the log records, first redoing the operations for committed transactions (to be sure their changes reached the file system data structures) and then undoing the operations for transactions that did not commit successfully before the crash. Periodically (usually every 5 seconds), a checkpoint record is written to the log. The system does not need log records prior to the checkpoint to recover from a crash. They can be discarded, so the log file does not grow without bounds. The first time after system startup that an NTFS volume is accessed, NTFS automatically performs file-system recovery.

This scheme does not guarantee that all the user-file contents are correct after a crash. It ensures only that the file-system data structures (the metadata files) are undamaged and reflect some consistent state that existed prior to the crash. It would be possible to extend the transaction scheme to cover user files, and Microsoft took some steps to do this in Windows Vista.

The log is stored in the third metadata file at the beginning of the volume. It is created with a fixed maximum size when the file system is formatted. It has two sections: the ***logging area,*** which is a circular queue of log records, and the ***restart area,*** which holds context information, such as the position in the logging area where NTFS should start reading during a recovery. In fact, the restart area holds two copies of its information, so recovery is still possible if one copy is damaged during the crash.

The logging functionality is provided by the ***log-file service.*** In addition to writing the log records and performing recovery actions, the log-file service keeps track of the free space in the log file. If the free space gets too low, the log-file service queues pending transactions, and NTFS halts all new I/O operations. After the in-progress operations complete, NTFS calls the cache manager to flush all data and then resets the log file and performs the queued transactions.

### 21.5.3 Security

The security of an NTFS volume is derived from the Windows object model. Each NTFS file references a security descriptor, which specifies the owner of the file, and an access-control list, which contains the access permissions granted or denied to each user or group listed. Early versions of NTFS used a separate security descriptor as an attribute of each file. Beginning with Windows 2000, the security-descriptor attribute points to a shared copy, with a significant savings in storage space and caching space; many, many files have identical security descriptors.

In normal operation, NTFS does not enforce permissions on traversal of directories in file path names. However, for compatibility with POSIX, these checks can be enabled. The latter option is inherently more expensive, since modern parsing of file path names uses prefix matching rather than directory-by-directory parsing of path names. Prefix matching is an algorithm that looks up strings in a cache and finds the entry with the longest match—for example, an entry for `\foo\bar\dir` would be a match for `\foo\bar\dir2\dir3\myfile`. The prefix-matching cache allows path-name traversal to begin much deeper in the tree, saving many steps. Enforcing traversal checks means that the user's access must be checked at each directory level. For instance, a user might lack permission to traverse `\foo\bar`, so starting at the access for `\foo\bar\dir` would be an error.

### 21.5.4 Compression

NTFS can perform data compression on individual files or on all data files in a directory. To compress a file, NTFS divides the file's data into **compression units**, which are blocks of 16 contiguous clusters. When a compression unit is written, a data-compression algorithm is applied. If the result fits into fewer than 16 clusters, the compressed version is stored. When reading, NTFS can determine whether data have been compressed: if they have been, the length of the stored compression unit is less than 16 clusters. To improve performance when reading contiguous compression units, NTFS prefetches and decompresses ahead of the application requests.

For sparse files or files that contain mostly zeros, NTFS uses another technique to save space. Clusters that contain only zeros because they have never been written are not actually allocated or stored on storage devices. Instead, gaps are left in the sequence of virtual-cluster numbers stored in the MFT entry for the file. When reading a file, if NTFS finds a gap in the virtual-cluster numbers, it just zero-fills that portion of the caller's buffer. This technique is also used by UNIX.

### 21.5.5 Mount Points, Symbolic Links, and Hard Links

Mount points are a form of symbolic link specific to directories on NTFS that were introduced in Windows 2000. They provide a mechanism for organizing storage volumes that is more flexible than the use of global names (like drive letters). A mount point is implemented as a symbolic link with associated data containing the true volume name. Ultimately, mount points will supplant drive letters completely, but there will be a long transition due to the dependence of many applications on the drive-letter scheme.

Windows Vista introduced support for a more general form of symbolic links, similar to those found in UNIX. The links can be absolute or relative, can point to objects that do not exist, and can point to both files and directories even across volumes. NTFS also supports **hard links**, where a single file has an entry in more than one directory of the same volume.

### 21.5.6 Change Journal

NTFS keeps a journal describing all changes that have been made to the file system. User-mode services can receive notifications of changes to the journal and then identify what files have changed by reading from the journal. The search indexer service uses the change journal to identify files that need to be re-indexed. The file-replication service uses it to identify files that need to be replicated across the network.

### 21.5.7 Volume Shadow Copies

Windows implements the capability of bringing a volume to a known state and then creating a shadow copy that can be used to back up a consistent view of the volume. This technique is known as ***snapshots*** in some other file systems. Making a shadow copy of a volume is a form of copy-on-write, where blocks modified after the shadow copy is created are stored in their original form in the copy. Achieving a consistent state for the volume requires the cooperation of applications, since the system cannot know when the data used by the application are in a stable state from which the application could be safely restarted.

The server version of Windows uses shadow copies to efficiently maintain old versions of files stored on file servers. This allows users to see documents as they existed at earlier points in time. A user can thus recover files that were accidentally deleted or simply look at a previous version of the file, all without pulling out backup media.

## 21.6 Networking

Windows supports both peer-to-peer and client–server networking. It also has facilities for network management. The networking components in Windows provide data transport, interprocess communication, file sharing across a network, and the ability to send print jobs to remote printers.

### 21.6.1 Network Interfaces

To describe networking in Windows, we must first mention two of the internal networking interfaces: the **Network Device Interface specificatio** (**NDIS**) and the **Transport Driver Interface** (**TDI**). The NDIS interface was developed in 1989 by Microsoft and 3Com to separate network adapters from transport protocols so that either could be changed without affecting the other. NDIS resides at the interface between the data-link and network layers in the ISO model and enables many protocols to operate over many different network adapters. In terms of the ISO model, the TDI is the interface between the transport layer (layer 4) and the session layer (layer 5). This interface enables any session-layer component to use any available transport mechanism. (Similar reasoning led to the streams mechanism in UNIX.) The TDI supports both connection-based and connectionless transport and has functions to send any type of data.

### 21.6.2 Protocols

Windows implements transport protocols as drivers. These drivers can be loaded and unloaded from the system dynamically, although in practice the system typically has to be rebooted after a change. Windows comes with several networking protocols. Next, we discuss a number of these protocols.

#### 21.6.2.1 Server Message Block

The **Server Message Block** (**SMB**) protocol was first introduced in MS-DOS 3.1. The system uses the protocol to send I/O requests over the network. The SMB protocol has four message types. `Session control` messages are commands that start and end a redirector connection to a shared resource at the server. A redirector uses `File` messages to access files at the server. `Printer` messages are used to send data to a remote print queue and to receive status information from the queue, and `Message` messages are used to communicate with another workstation. A version of the SMB protocol was published as the **Common Internet File System** (**CIFS**) and is supported on a number of operating systems.

#### 21.6.2.2 Transmission Control Protocol/Internet Protocol

The transmission control protocol/Internet protocol (TCP/IP) suite that is used on the Internet has become the de facto standard networking infrastructure. Windows uses TCP/IP to connect to a wide variety of operating systems and hardware platforms. The Windows TCP/IP package includes the simple network-management protocol (SNMP), the dynamic host-configuration protocol (DHCP), and the older Windows Internet name service (WINS). Windows Vista introduced a new implementation of TCP/IP that supports both `IPv4` and `IPv6` in the same network stack. This new implementation also supports offloading of the network stack onto advanced hardware to achieve very high performance for servers.

Windows provides a software firewall that limits the TCP ports that can be used by programs for network communication. Network firewalls are commonly implemented in routers and are a very important security measure. Having a firewall built into the operating system makes a hardware router unnecessary, and it also provides more integrated management and easier use.

#### 21.6.2.3 Point-to-Point Tunneling Protocol

The **Point-to-Point Tunneling Protocol** (**PPTP**) is a protocol provided by Windows to communicate between remote-access server modules running on Windows server machines and other client systems that are connected over the Internet. The remote-access servers can encrypt data sent over the connection, and they support multiprotocol **virtual private networks** (**VPNs**) on the Internet.

#### 21.6.2.4 HTTP Protocol

The HTTP protocol is used to `get/put` information using the World Wide Web. Windows implements HTTP using a kernel-mode driver, so web servers can operate with a low-overhead connection to the networking stack. HTTP is a fairly general protocol that Windows makes available as a transport option for implementing RPC.

#### 21.6.2.5 Web-Distributed Authoring and Versioning Protocol

Web-distributed authoring and versioning (WebDAV) is an HTTP-based protocol for collaborative authoring across a network. Windows builds a WebDAV redirector into the file system. Being built directly into the file system enables WebDAV to work with other file-system features, such as encryption. Personal files can then be stored securely in a public place. Because WebDAV uses HTTP, which is a `get/put` protocol, Windows has to cache the files locally so programs can use `read` and `write` operations on parts of the files.

#### 21.6.2.6 Named Pipes

**Named pipes** are a connection-oriented messaging mechanism. A process can use named pipes to communicate with other processes on the same machine. Since named pipes are accessed through the file-system interface, the security mechanisms used for file objects also apply to named pipes. The SMB protocol supports named pipes, so named pipes can also be used for communication between processes on different systems.

The format of pipe names follows the **Uniform Naming Convention** (**UNC**). A UNC name looks like a typical remote file name. The format is `\\server_name\share_name\x\y\z`, where `server_name` identifies a server on the network; `share_name` identifies any resource that is made available to network users, such as directories, files, named pipes, and printers; and `\x\y\z` is a normal file path name.

#### 21.6.2.7 Remote Procedure Calls

Remote procedure calls (RPCs), mentioned earlier, are client–server mechanisms that enable an application on one machine to make a procedure call to code on another machine. The client calls a local procedure—a stub routine—which packs its arguments into a message and sends them across the network to a particular server process. The client-side stub routine then blocks. Meanwhile, the server unpacks the message, calls the procedure, packs the return results into a message, and sends them back to the client stub. The client stub unblocks, receives the message, unpacks the results of the RPC, and returns them to the caller. This packing of arguments is sometimes called **marshaling**. The client stub code and the descriptors necessary to pack and unpack the arguments for an RPC are compiled from a specification written in the **Microsoft Interface Definition Language**.

The Windows RPC mechanism follows the widely used distributed-computing-environment standard for RPC messages, so programs written to use Windows RPCs are highly portable. The RPC standard is detailed. It hides many of the architectural differences among computers, such as the sizes of binary numbers and the order of bytes and bits in computer words, by specifying standard data formats for RPC messages.

#### 21.6.2.8 Component Object Model

The **Component Object Model** (**COM**) is a mechanism for interprocess communication that was developed for Windows. A COM object provides a well-defined interface to manipulate the data in the object. For instance, COM is the infrastructure used by Microsoft's **Object Linking and Embedding** (**OLE**) technology for inserting spreadsheets into Microsoft Word documents. Many Windows services provide COM interfaces. In addition, a distributed extension called **DCOM** can be used over a network utilizing RPC to provide a transparent method of developing distributed applications.

### 21.6.3 Redirectors and Servers

In Windows, an application can use the Windows I/O API to access files from a remote computer as though they were local, provided that the remote computer is running a CIFS server such as those provided by Windows. A **redirector** is the client-side object that forwards I/O requests to a remote system, where they are satisfied by a server. For performance and security, the redirectors and servers run in kernel mode.

In more detail, access to a remote file occurs as follows:

1. The application calls the I/O manager to request that a file be opened with a file name in the standard UNC format.
2. The I/O manager builds an I/O request packet, as described in Section 21.3.5.5.
3. The I/O manager recognizes that the access is for a remote file and calls a driver called a **Multiple UNC Provider** (**MUP**).

4. The MUP sends the I/O request packet asynchronously to all registered redirectors.
5. A redirector that can satisfy the request responds to the MUP. To avoid asking all the redirectors the same question in the future, the MUP uses a cache to remember which redirector can handle this file.
6. The redirector sends the network request to the remote system.
7. The remote-system network drivers receive the request and pass it to the server driver.
8. The server driver hands the request to the proper local file-system driver.
9. The proper device driver is called to access the data.
10. The results are returned to the server driver, which sends the data back to the requesting redirector. The redirector then returns the data to the calling application via the I/O manager.

A similar process occurs for applications that use the Win32 network API, rather than the UNC services, except that a module called a ***multi-provider router*** is used instead of a MUP.

For portability, redirectors and servers use the TDI API for network transport. The requests themselves are expressed in a higher-level protocol, which by default is the SMB protocol described in Section 21.6.2. The list of redirectors is maintained in the system hive of the registry.

### 21.6.3.1 Distributed File System

UNC names are not always convenient, because multiple file servers may be available to serve the same content and UNC names explicitly include the name of the server. Windows supports a **distributed file-system** (**DFS**) protocol that allows a network administrator to serve up files from multiple servers using a single distributed name space.

### 21.6.3.2 Folder Redirection and Client-Side Caching

To improve the PC experience for users who frequently switch among computers, Windows allows administrators to give users **roaming profiles**, which keep users' preferences and other settings on servers. **Folder redirection** is then used to automatically store a user's documents and other files on a server.

This works well until one of the computers is no longer attached to the network, as when a user takes a laptop onto an airplane. To give users off-line access to their redirected files, Windows uses **client-side caching** (**CSC**). CSC is also used when the computer is on-line to keep copies of the server files on the local machine for better performance. The files are pushed up to the server as they are changed. If the computer becomes disconnected, the files are still available, and the update of the server is deferred until the next time the computer is online.

### 21.6.4 Domains

Many networked environments have natural groups of users, such as students in a computer laboratory at school or employees in one department in a business. Frequently, we want all the members of the group to be able to access shared resources on their various computers in the group. To manage the global access rights within such groups, Windows uses the concept of a domain. Previously, these domains had no relationship whatsoever to the domain-name system (DNS) that maps Internet host names to IP addresses. Now, however, they are closely related.

Specifically, a Windows domain is a group of Windows workstations and servers that share a common security policy and user database. Since Windows uses the Kerberos protocol for trust and authentication, a Windows domain is the same thing as a Kerberos realm. Windows uses a hierarchical approach for establishing trust relationships between related domains. The trust relationships are based on DNS and allow transitive trusts that can flow up and down the hierarchy. This approach reduces the number of trusts required for $n$ domains from $n * (n - 1)$ to $O(n)$. The workstations in the domain trust the domain controller to give correct information about the access rights of each user (loaded into the user's access token by `lsaas`). All users retain the ability to restrict access to their own workstations, however, no matter what any domain controller may say.

### 21.6.5 Active Directory

**Active Directory** is the Windows implementation of **Lightweight Directory-Access Protocol** (**LDAP**) services. Active Directory stores the topology information about the domain, keeps the domain-based user and group accounts and passwords, and provides a domain-based store for Windows features that need it, such as **Windows group policy**. Administrators use group policies to establish uniform standards for desktop preferences and software. For many corporate information-technology groups, uniformity drastically reduces the cost of computing.

## 21.7 Programmer Interface

The **Win32 API** is the fundamental interface to the capabilities of Windows. This section describes five main aspects of the Win32 API: access to kernel objects, sharing of objects between processes, process management, interprocess communication, and memory management.

### 21.7.1 Access to Kernel Objects

The Windows kernel provides many services that application programs can use. Application programs obtain these services by manipulating kernel objects. A process gains access to a kernel object named `XXX` by calling the `CreateXXX` function to open a handle to an instance of `XXX`. This handle is unique to the process. Depending on which object is being opened, if the `Create()` function fails, it may return 0, or it may return a special constant named `INVALID_HANDLE_VALUE`. A process can close any handle by calling the

```
SECURITY_ATTRIBUTES sa;
sa.nlength = sizeof(sa);
sa.lpSecurityDescriptor = NULL;
sa.bInheritHandle = TRUE;
HANDLE hSemaphore = CreateSemaphore(&sa, 1, 1, NULL);
WCHAR wszCommandline[MAX_PATH];
StringCchPrintf(wszCommandLine, _countof(wszCommandLine),
   L"another_process.exe %d", hSemaphore);
CreateProcess(L"another_process.exe", wszCommandline,
   NULL, NULL, TRUE, . . .);
```

**Figure 21.7** Code enabling a child to share an object by inheriting a handle.

`CloseHandle()` function, and the system may delete the object if the count of handles referencing the object in all processes drops to zero.

### 21.7.2 Sharing Objects Between Processes

Windows provides three ways to share objects between processes. The first way is for a child process to inherit a handle to the object. When the parent calls the `CreateXXX` function, the parent supplies a `SECURITIES_ATTRIBUTES` structure with the `bInheritHandle` field set to `TRUE`. This field creates an inheritable handle. Next, the child process is created, passing a value of `TRUE` to the `CreateProcess()` function's `bInheritHandle` argument. Figure 21.7 shows a code sample that creates a semaphore handle inherited by a child process.

Assuming the child process knows which handles are shared, the parent and child can achieve interprocess communication through the shared objects. In the example in Figure 21.7, the child process gets the value of the handle from the first command-line argument and then shares the semaphore with the parent process.

The second way to share objects is for one process to give the object a name when the object is created and for the second process to open the name. This method has two drawbacks: Windows does not provide a way to check whether an object with the chosen name already exists, and the object name space is global, without regard to the object type. For instance, two applications may create and share a single object named "foo" when two distinct objects—possibly of different types—were desired.

Named objects have the advantage that unrelated processes can readily share them. The first process calls one of the `CreateXXX` functions and supplies a name as a parameter. The second process gets a handle to share the object by calling `OpenXXX()` (or `CreateXXX`) with the same name, as shown in the example in Figure 21.8.

The third way to share objects is via the `DuplicateHandle()` function. This method requires some other method of interprocess communication to pass the duplicated handle. Given a handle to a process and the value of a handle within that process, a second process can get a handle to the same object and thus share it. An example of this method is shown in Figure 21.9.

```
// Process A
. . .
HANDLE hSemaphore = CreateSemaphore(NULL, 1, 1, L"MySEM1");
. . .

// Process B
. . .
HANDLE hSemaphore = OpenSemaphore(SEMAPHORE_ALL_ACCESS,
   FALSE, L"MySEM1");
. . .
```

**Figure 21.8** Code for sharing an object by name lookup.

### 21.7.3 Process Management

In Windows, a **process** is a loaded instance of an application and a **thread** is an executable unit of code that can be scheduled by the kernel dispatcher. Thus, a process contains one or more threads. A process is created when a thread in some other process calls the `CreateProcess()` API. This routine loads any dynamic link libraries used by the process and creates an initial thread in the process. Additional threads can be created by the `CreateThread()` function.

```
// Process A wants to give Process B access to a semaphore

// Process A

DWORD dwProcessBId; // must; from some IPC mechanism
HANDLE hSemaphore = CreateSemaphore(NULL, 1, 1, NULL);
HANDLE hProcess = OpenProcess(PROCESS_DUP_HANDLE, FALSE,
   dwProcessBId);
HANDLE hSemaphoreCopy;
DuplicateHandle(GetCurrentProcess(), hSemaphore,
   hProcess, &hSemaphoreCopy,
   0, FALSE, DUPLICATE_SAME_ACCESS);
// send the value of the semaphore to Process B
// using a message or shared memory object
. . .

// Process B
HANDLE hSemaphore = // value of semaphore from message
// use hSemaphore to access the semaphore
. . .
```

**Figure 21.9** Code for sharing an object by passing a handle.

Each thread is created with its own stack, which defaults to 1 MB unless otherwise specified in an argument to `CreateThread()`.

#### 21.7.3.1 Scheduling Rule

Priorities in the Win32 environment are based on the native kernel (NT) scheduling model, but not all priority values may be chosen. The Win32 API uses six priority classes:

1. `IDLE_PRIORITY_CLASS` (NT priority level 4)
2. `BELOW_NORMAL_PRIORITY_CLASS` (NT priority level 6)
3. `NORMAL_PRIORITY_CLASS` (NT priority level 8)
4. `ABOVE_NORMAL_PRIORITY_CLASS` (NT priority level 10)
5. `HIGH_PRIORITY_CLASS` (NT priority level 13)
6. `REALTIME_PRIORITY_CLASS` (NT priority level 24)

Processes are typically members of the `NORMAL_PRIORITY_CLASS` unless the parent of the process was of the `IDLE_PRIORITY_CLASS` or another class was specified when `CreateProcess` was called. The priority class of a process is the default for all threads that execute in the process. It can be changed with the `SetPriorityClass()` function or by passing an argument to the `start` command. Only users with the ***increase scheduling priority*** privilege can move a process into the `REALTIME_PRIORITY_CLASS`. Administrators and power users have this privilege by default.

When a user is switching between interactive processes and workloads, the system needs to schedule the appropriate threads so as to provide good responsiveness, which leads to a shorter quantums of execution. Yet, once the user has chosen a particular process, a good amount of throughput from this particular process is also expected. For this reason, Windows has a special scheduling rule for processes not in the `REALTIME_PRIORITY_CLASS`. Windows distinguishes between the process associated with the foreground window on the screen and the other (background) processes. When a process moves into the foreground, Windows increases the scheduling quantum for all its threads by a factor of 3; CPU-bound threads in the foreground process will run three times longer than similar threads in background processes. Because server systems always operate with a much larger quantum than client systems—a factor of 6—this behavior is not enabled for server systems. For both types of systems, however, the scheduling parameters can be customized through the appropriate system dialog or registry key.

#### 21.7.3.2 Thread Priorities

A thread starts with an initial priority determined by its class. The priority can be altered by the `SetThreadPriority()` function. This function takes an argument that specifies a priority relative to the base priority of its class:

- `THREAD_PRIORITY_LOWEST`: base – 2
- `THREAD_PRIORITY_BELOW_NORMAL`: base – 1

- `THREAD_PRIORITY_NORMAL`: base + 0
- `THREAD_PRIORITY_ABOVE_NORMAL`: base + 1
- `THREAD_PRIORITY_HIGHEST`: base + 2

Two other designations are also used to adjust the priority. Recall from Section 21.3.4.3 that the kernel has two priority classes: 16–31 for the static class and 1–15 for the variable class. `THREAD_PRIORITY_IDLE` sets the priority to 16 for static-priority threads and to 1 for variable-priority threads. `THREAD_PRIORITY_TIME_CRITICAL` sets the priority to 31 for real-time threads and to 15 for variable-priority threads.

The kernel adjusts the priority of a variable class thread dynamically depending on whether the thread is I/O bound or CPU bound. The Win32 API provides a method to disable this adjustment via `SetProcessPriorityBoost()` and `SetThreadPriorityBoost()` functions.

### 21.7.3.3 Thread Suspend and Resume

A thread can be created in a ***suspended state*** or can be placed in a suspended state later by use of the `SuspendThread()` function. Before a suspended thread can be scheduled by the kernel dispatcher, it must be moved out of the suspended state by use of the `ResumeThread()` function. Both functions set a counter so that if a thread is suspended twice, it must be resumed twice before it can run.

### 21.7.3.4 Thread Synchronization

To synchronize concurrent access to shared objects by threads, the kernel provides synchronization objects, such as semaphores and mutexes. These are dispatcher objects, as discussed in Section 21.3.4.3. Threads can also synchronize with kernel services operating on kernel objects—such as threads, processes, and files—because these are also dispatcher objects. Synchronization with kernel dispatcher objects can be achieved by use of the `WaitForSingleObject()` and `WaitForMultipleObjects()` functions; these functions wait for one or more dispatcher objects to be signaled.

Another method of synchronization is available to threads within the same process that want to execute code exclusively. The Win32 **critical section object** is a user-mode mutex object that can often be acquired and released without entering the kernel. On a multiprocessor, a Win32 critical section will attempt to spin while waiting for a critical section held by another thread to be released. If the spinning takes too long, the acquiring thread will allocate a kernel mutex and yield its CPU. Critical sections are particularly efficient because the kernel mutex is allocated only when there is contention and then used only after attempting to spin. Most mutexes in programs are never actually contended, so the savings are significant.

Before using a critical section, some thread in the process must call `InitializeCriticalSection()`. Each thread that wants to acquire the mutex calls `EnterCriticalSection()` and then later calls `LeaveCriticalSection()` to release the mutex. There is also a `TryEnterCriticalSection()` function, which attempts to acquire the mutex without blocking.

For programs that want user-mode reader–writer locks rather than mutexes, Win32 supports **slim reader–writer (SRW) locks**. SRW locks have APIs similar to those for critical sections, such as `InitializeSRWLock`, `AcquireSRWLockXXX`, and `ReleaseSRWLockXXX`, where `XXX` is either `Exclusive` or `Shared`, depending on whether the thread wants write access or only read access to the object protected by the lock. The Win32 API also supports **condition variables**, which can be used with either critical sections or SRW locks.

#### 21.7.3.5 Thread Pool

Repeatedly creating and deleting threads can be expensive for applications and services that perform small amounts of work in each instantiation. The Win32 thread pool provides user-mode programs with three services: a queue to which work requests may be submitted (via the `SubmitThreadpoolWork()` function), an API that can be used to bind callbacks to waitable handles (`RegisterWaitForSingleObject()`), and APIs to work with timers (`CreateThreadpoolTimer()` and `WaitForThreadpoolTimerCallbacks()`) and to bind callbacks to I/O completion queues (`BindIoCompletionCallback()`).

The goal of using a thread pool is to increase performance and reduce memory footprint. Threads are relatively expensive, and each processor can be executing only one thread at a time no matter how many threads are available. The thread pool attempts to reduce the number of runnable threads by slightly delaying work requests (reusing each thread for many requests) while providing enough threads to effectively utilize the machine's CPUs. The wait and I/O- and timer-callback APIs allow the thread pool to further reduce the number of threads in a process, using far fewer threads than would be necessary if a process were to devote separate threads to servicing each waitable handle, timer, or completion port.

#### 21.7.3.6 Fibers

A **fiber** is user-mode code that is scheduled according to a user-defined scheduling algorithm. Fibers are completely a user-mode facility; the kernel is not aware that they exist. The fiber mechanism uses Windows threads as if they were CPUs to execute the fibers. Fibers are cooperatively scheduled, meaning that they are never preempted but must explicitly yield the thread on which they are running. When a fiber yields a thread, another fiber can be scheduled on it by the run-time system (the programming language run-time code).

The system creates a fiber by calling either `ConvertThreadToFiber()` or `CreateFiber()`. The primary difference between these functions is that `CreateFiber()` does not begin executing the fiber that was created. To begin execution, the application must call `SwitchToFiber()`. The application can terminate a fiber by calling `DeleteFiber()`.

Fibers are not recommended for threads that use Win32 APIs rather than standard C-library functions because of potential incompatibilities. Win32 user-mode threads have a **thread-environment block (TEB)** that contains numerous per-thread fields used by the Win32 APIs. Fibers must share the TEB of the thread on which they are running. This can lead to problems when a Win32 interface puts state information into the TEB for one fiber and then the information is overwritten by a different fiber. Fibers are included in the Win32 API to facilitate the porting of legacy UNIX applications that were written for a user-mode thread model such as Pthreads.

#### 21.7.3.7 User-Mode Scheduling UMS and ConcRT

A new mechanism in Windows 7, user-mode scheduling (UMS), addressed several limitations of fibers. As just noted, fibers are unreliable for executing Win32 APIs because they do not have their own TEBs. When a thread running a fiber blocks in the kernel, the user scheduler loses control of the CPU for a time as the kernel dispatcher takes over scheduling. Problems may result when fibers change the kernel state of a thread, such as the priority or impersonation token, or when they start asynchronous I/O.

UMS provides an alternative model by recognizing that each Windows thread is actually two threads: a kernel thread (KT) and a user thread (UT). Each type of thread has its own stack and its own set of saved registers. The KT and UT appear as a single thread to the programmer because UTs can never block but must always enter the kernel, where an implicit switch to the corresponding KT takes place. UMS uses each UT's TEB to uniquely identify the UT. When a UT enters the kernel, an explicit switch is made to the KT that corresponds to the UT identified by the current TEB. The reason the kernel does not know which UT is running is that UTs can invoke a user-mode scheduler, as fibers do. But in UMS, the scheduler switches UTs, including switching the TEBs.

When a UT enters the kernel, its KT may block. When this happens, the kernel switches to a scheduling thread, which UMS calls a ***primary thread***, and uses this thread to reenter the user-mode scheduler so that it can pick another UT to run. Eventually, a blocked KT will complete its operation and be ready to return to user mode. Since UMS has already reentered the user-mode scheduler to run a different UT, UMS queues the UT corresponding to the completed KT to a completion list in user mode. When the user-mode scheduler is choosing a new UT to switch to, it can examine the completion list and treat any UT on the list as a candidate for scheduling. The key features of UMS are depicted in Figure 21.10.

Unlike fibers, UMS is not intended to be used directly by programmers. The details of writing user-mode schedulers can be very challenging, and UMS does not include such a scheduler. Rather, the schedulers come from programming language libraries that build on top of UMS. Microsoft Visual Studio 2010 shipped with Concurrency Runtime (ConcRT), a concurrent programming framework for C++. ConcRT provides a user-mode scheduler together with facilities for decomposing programs into tasks, which can then be scheduled on the available CPUs. ConcRT provides support for `par_for` styles of constructs, as well as rudimentary resource management and task synchronization primitives. However, as of Visual Studio 2013, the UMS scheduling mode is no longer available in ConcRT. Significant performance metrics showed that true parallel programs that are well written do not spend a large amount of time context-switching between their tasks. The benefits that UMS provided in this space did not outweigh the complexity of maintaining a separate scheduler—in some cases, even the default NT scheduler performed better.

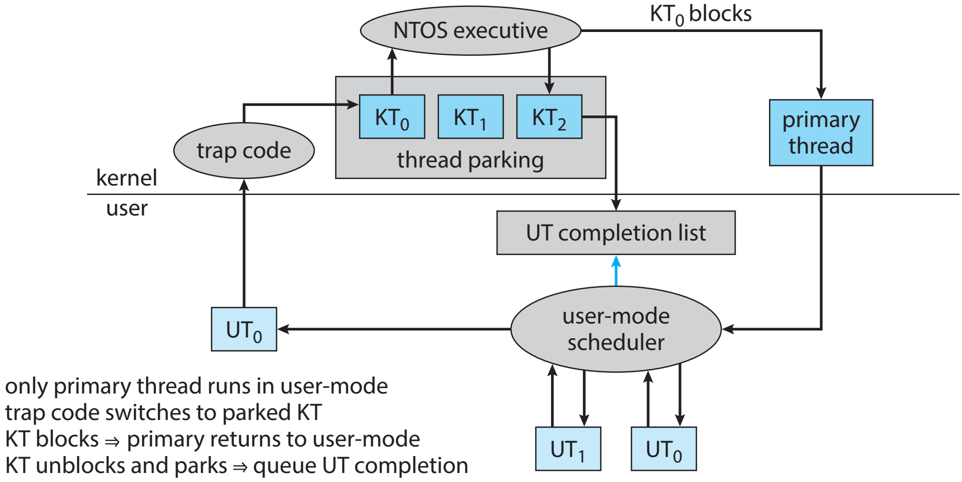

KT$_0$ blocks
NTOS executive
KT$_0$
KT$_1$
KT$_2$
thread parking
trap code
primary thread
kernel
user
UT completion list
UT$_0$
user-mode scheduler
UT$_1$
UT$_0$
only primary thread runs in user-mode
trap code switches to parked KT
KT blocks ⇒ primary returns to user-mode
KT unblocks and parks ⇒ queue UT completion

**Figure 21.10** User-mode scheduling.

#### 21.7.3.8 Winsock

**Winsock** is the Windows sockets API. Winsock is a session-layer interface that is largely compatible with BSD sockets but has some added Windows extensions. It provides a standardized interface to many transport protocols that may have different addressing schemes, so that any Winsock application can run on any Winsock-compliant protocol stack. Winsock underwent a major update in Windows Vista to add tracing, `IPv6` support, impersonation, new security APIs, and many other features.

Winsock follows the Windows Open System Architecture (WOSA) model, which provides a standard service provider interface (SPI) between applications and networking protocols. Applications can load and unload ***layered protocols*** that build additional functionality, such as additional security, on top of the transport protocol layers. Winsock supports asynchronous operations and notifications, reliable multicasting, secure sockets, and kernel mode sockets. It also supports simpler usage models, like the `WSAConnectByName()` function, which accepts the target as strings specifying the name or IP address of the server and the service or port number of the destination port.

### 21.7.4 IPC Using Windows Messaging

Win32 applications handle interprocess communication in several ways. The typical high-performance way is by using local RPCs or named pipes. Another is by using shared kernel objects, such as named section objects, and a synchronization object, such as an event. Yet another is by using the Windows messaging facility—an approach that is particularly popular for Win32 GUI applications. One thread can send a message to another thread or to a window by calling `PostMessage()`, `PostThreadMessage()`, `SendMessage()`, `SendThreadMessage()`, or `SendMessageCallback()`. ***Posting*** a message and ***sending*** a message differ in this way: The post routines are asynchronous; they return immediately, and the calling thread does not know when the message is actually delivered. The send routines are synchronous; they block the caller until the message has been delivered and processed.

In addition to sending a message, a thread can send data with the message. Since processes have separate address spaces, the data must be copied. The system copies data by calling `SendMessage()` to send a message of type `WM_COPYDATA` with a `COPYDATASTRUCT` data structure that contains the length and address of the data to be transferred. When the message is sent, Windows copies the data to a new block of memory and gives the virtual address of the new block to the receiving process.

Every Win32 GUI thread has its own input queue from which it receives messages. If a Win32 application does not call `GetMessage()` to handle events on its input queue, the queue fills up; and after about five seconds, the task manager marks the application as "Not Responding." Note that message passing is subject to the integrity level mechanism introduced earlier. Thus, a process may not send a message such as WM_COPYDATA to a process with a higher integrity level, unless a special Windows API is used to remove the protection (ChangeWindowMessageFilterEx).

## 21.7.5 Memory Management

The Win32 API provides several ways for an application to use memory: virtual memory, memory-mapped files, heaps, thread-local storage, and AWE physical memory.

### 21.7.5.1 Virtual Memory

An application calls `VirtualAlloc()` to reserve or commit virtual memory and `VirtualFree()` to de-commit or release the memory. These functions enable the application to specify the virtual address at which the memory is allocated. (Otherwise, a random address is selected, which is recommended for security reasons.) The functions operate on multiples of the memory page size but, for historical reasons, always return memory allocated on a 64-KB boundary. Examples of these functions appear in Figure 21.11. The `VirtualAllocEx()` and `VirtualFreeEx()` functions can be used to allocate and free memory in a separate process, while `VirtualAllocExNuma()` can be used to leverage memory locality on NUMA systems.

### 21.7.5.2 Memory-Mapped Files

Another way for an application to use memory is by memory-mapping a file into its address space. Memory mapping is also a convenient way for two processes to share memory: both processes map the same file into their virtual memory. Memory mapping is a multistage process, as you can see in the example in Figure 21.12.

If a process wants to map some address space just to share a memory region with another process, no file is needed. The process calls `CreateFileMapping()` with a file handle of `0xffffffff`, a particular size, and (optionally) a name. The resulting file-mapping object can be shared by inheritance, by name lookup (if it was named), or by handle duplication.

```
// reserve 16 MB at the top of our address space
PVOID pBuf = VirtualAlloc(NULL, 0x1000000,
    MEM_RESERVE | MEM_TOP_DOWN, PAGE_READWRITE);
// commit the upper 8 MB of the allocated space
VirtualAlloc((LPVOID)((DWORD_PTR)pBuf + 0x800000), 0x800000,
    MEM_COMMIT, PAGE_READWRITE);
// do something with the memory
. . .
// now decommit the memory
VirtualFree((LPVOID)((DWORD_PTR)pBuf + 0x800000), 0x800000,
    MEM_DECOMMIT);
// release all of the allocated address space
VirtualFree(pBuf, 0, MEM_RELEASE);
```

**Figure 21.11** Code fragments for allocating virtual memory.

#### 21.7.5.3 Heaps

Heaps provide a third way for applications to use memory, just as with `malloc()` and `free()` in standard C or `new()` and `delete()` in C++. A heap in the Win32 environment is a region of pre-committed address space. When a Win32 process is initialized, it is created with a **default heap**. Since most Win32

```
// set the file mapping size to 8MB
DWORD dwSize = 0x800000;
// open the file or create it if it does not exist
HANDLE hFile = CreateFile(L"somefile.ext",
   GENERIC_READ | GENERIC_WRITE,
   FILE_SHARE_READ | FILE_SHARE_WRITE, NULL,
   OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
// create the file mapping
HANDLE hMap = CreateFileMapping(hFile,
   PAGE_READWRITE | SEC_COMMIT, 0, dwSize, L"SHM_1");
// now get a view of the space mapped
PVOID pBuf = MapViewOfFile(hMap, FILE_MAP_ALL_ACCESS,
   0, 0, 0, dwSize);
// do something with the mapped file
. . .
// now unmap the file
UnmapViewOfFile(pBuf);
CloseHandle(hMap);
CloseHandle(hFile);
```

**Figure 21.12** Code fragments for memory mapping of a file.

applications are multithreaded, access to the heap is synchronized to protect the heap's space-allocation data structures from being damaged by concurrent updates by multiple threads. The advantage of the heap is that it can be used to make allocations as small as 1 byte, because the underlying memory pages have already been committed. Unfortunately, heap memory cannot be shared or marked as read-only, because all heap allocations share the same pages. However, by using `HeapCreate()`, a programmer can create his or her own heap, which can be marked as read-only with `HeapProtect()`, created as an executable heap, or even allocated on a specific NUMA node.

Win32 provides several heap-management functions so that a process can allocate and manage a private heap. These functions are `HeapCreate()`, `HeapAlloc()`, `HeapRealloc()`, `HeapSize()`, `HeapFree()`, and `HeapDestroy()`. The Win32 API also provides the `HeapLock()` and `HeapUnlock()` functions to enable a thread to gain exclusive access to a heap. Note that these functions perform only synchronization; they do not truly "lock" pages against malicious or buggy code that bypasses the heap layer.

The original Win32 heap was optimized for efficient use of space. This led to significant problems with fragmentation of the address space for larger server programs that ran for long periods of time. A new **low-fragmentation heap** (**LFH**) design introduced in Windows XP greatly reduced the fragmentation problem. The heap manager in Windows 7 and later versions automatically turns on LFH as appropriate. Additionally, the heap is a primary target of attackers using vulnerabilities such as double-free, use-after-free, and other memory-corruption-related attacks. Each version of Windows, including Windows 10, has added more randomness, entropy, and security mitigations to prevent attackers from guessing the ordering, size, location, and content of heap allocations.

### 21.7.5.4 Thread-Local Storage

A fourth way for applications to use memory is through a **thread-local storage** (**TLS**) mechanism. Functions that rely on global or static data typically fail to work properly in a multithreaded environment. For instance, the C run-time function `strtok()` uses a static variable to keep track of its current position while parsing a string. For two concurrent threads to execute `strtok()` correctly, they need separate `current position` variables. TLS provides a way to maintain instances of variables that are global to the function being executed but not shared with any other thread.

TLS provides both dynamic and static methods of creating thread-local storage. The dynamic method is illustrated in Figure 21.13. The TLS mechanism allocates global heap storage and attaches it to the thread environment block (TEB) that Windows allocates to every user-mode thread. The TEB is readily accessible by each thread and is used not just for TLS but for all the per-thread state information in user mode.

### 21.7.5.5 AWE Memory

A final way for applications to use memory is through the **Address Windowing Extension** (**AWE**) functionality. This mechanism allows a developer to directly

```
// reserve a slot for a variable
DWORD dwVarIndex = T1sAlloc();
// make sure a slot was available
if (dwVarIndex == TLS_OUT_OF_INDEXES)
return;
// set it to the value 10
T1sSetValue(dwVarIndex, (LPVOID)10);
// get the value
DWORD dwVar = (DWORD)(DWORD_PTR)T1sGetValue(dwVarIndex);
// release the index
T1sFree(dwVarIndex);
```

**Figure 21.13** Code for dynamic thread-local storage.

request free physical pages of RAM from the memory manager (through `AllocateUserPhysicalPages()`) and later commit virtual memory on top of the physical pages using `VirtualAlloc()`. By requesting various regions of physical memory (including scatter-gather support), a user-mode application can access more physical memory than virtual address space; this is useful on 32-bit systems, which may have more than 4 GB of RAM). In addition, the application can bypass the memory manager's caching, paging, and coloring algorithms. Similar to UMS, AWE may thus offer a way for certain applications to extract additional performance or customization beyond what Windows offers by default. SQL Server, for example, uses AWE memory.

To use a thread-local static variable, the application declares the variable as follows to ensure that every thread has its own private copy:

```
__declspec(thread) DWORD cur_pos = 0;
```

## 21.8 Summary

- Microsoft designed Windows to be an extensible, portable operating system—one able to take advantage of new techniques and hardware.
- Windows supports multiple operating environments and symmetric multiprocessing, including both 32-bit and 64-bit processors and NUMA computers.
- The use of kernel objects to provide basic services, along with support for client–server computing, enables Windows to support a wide variety of application environments.
- Windows provides virtual memory, integrated caching, and preemptive scheduling.

- To protect user data and guarantee program integrity, Windows supports elaborate security mechanisms and exploit mitigations and takes advantage of hardware virtualization.
- Windows runs on a wide variety of computers, so users can choose and upgrade hardware to match their budgets and performance requirements without needing to alter the applications they run.
- By including internationalization features, Windows can run in a variety of countries and many languages.
- Windows has sophisticated scheduling and memory-management algorithms for performance and scalability.
- Recent versions of Windows have added power management and fast sleep and wake features, and decreased resource use in several areas to be more useful on mobile systems such as phones and tablets.
- The Windows volume manager and NTFS file system provide a sophisticated set of features for desktop as well as server systems.
- The Win32 API programming environment is feature rich and expansive, allowing programmers to use all of Windows's features in their programs.

## Practice Exercises

**21.1** What type of operating system is Windows? Describe two of its major features.

**21.2** List the design goals of Windows. Describe two in detail.

**21.3** Describe the booting process for a Windows system.

**21.4** Describe the three main architectural layers of the Windows kernel.

**21.5** What is the job of the object manager?

**21.6** What types of services does the process manager provide?

**21.7** What is a local procedure call?

**21.8** What are the responsibilities of the I/O manager?

**21.9** What types of networking does Windows support? How does Windows implement transport protocols? Describe two networking protocols.

**21.10** How is the NTFS namespace organized?

**21.11** How does NTFS handle data structures? How does NTFS recover from a system crash? What is guaranteed after a recovery takes place?

**21.12** How does Windows allocate user memory?

**21.13** Describe some of the ways in which an application can use memory via the Win32 API.

## Further Reading

[Russinovich et al. (2017)] give a deep overview of Windows 10 and considerable technical detail about system internals and components.

## Bibliography

**[Russinovich et al. (2017)]** M. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals–Part 1,* Seventh Edition, Microsoft Press (2017).

## Chapter 21 Exercises

**21.14** Under what circumstances would one use the deferred procedure calls facility in Windows?

**21.15** What is a handle, and how does a process obtain a handle?

**21.16** Describe the management scheme of the virtual memory manager. How does the VM manager improve performance?

**21.17** Describe a useful application of the no-access page facility provided in Windows.

**21.18** Describe the three techniques used for communicating data in a local procedure call. What settings are most conducive to the application of the different message-passing techniques?

**21.19** What manages caching in Windows? How is the cache managed?

**21.20** How does the NTFS directory structure differ from the directory structure used in UNIX operating systems?

**21.21** What is a process, and how is it managed in Windows?

**21.22** What is the fiber abstraction provided by Windows? How does it differ from the thread abstraction?

**21.23** How does user-mode scheduling (UMS) in Windows 7 differ from fibers? What are some trade-offs between fibers and UMS?

**21.24** UMS considers a thread to have two parts, a UT and a KT. How might it be useful to allow UTs to continue executing in parallel with their KTs?

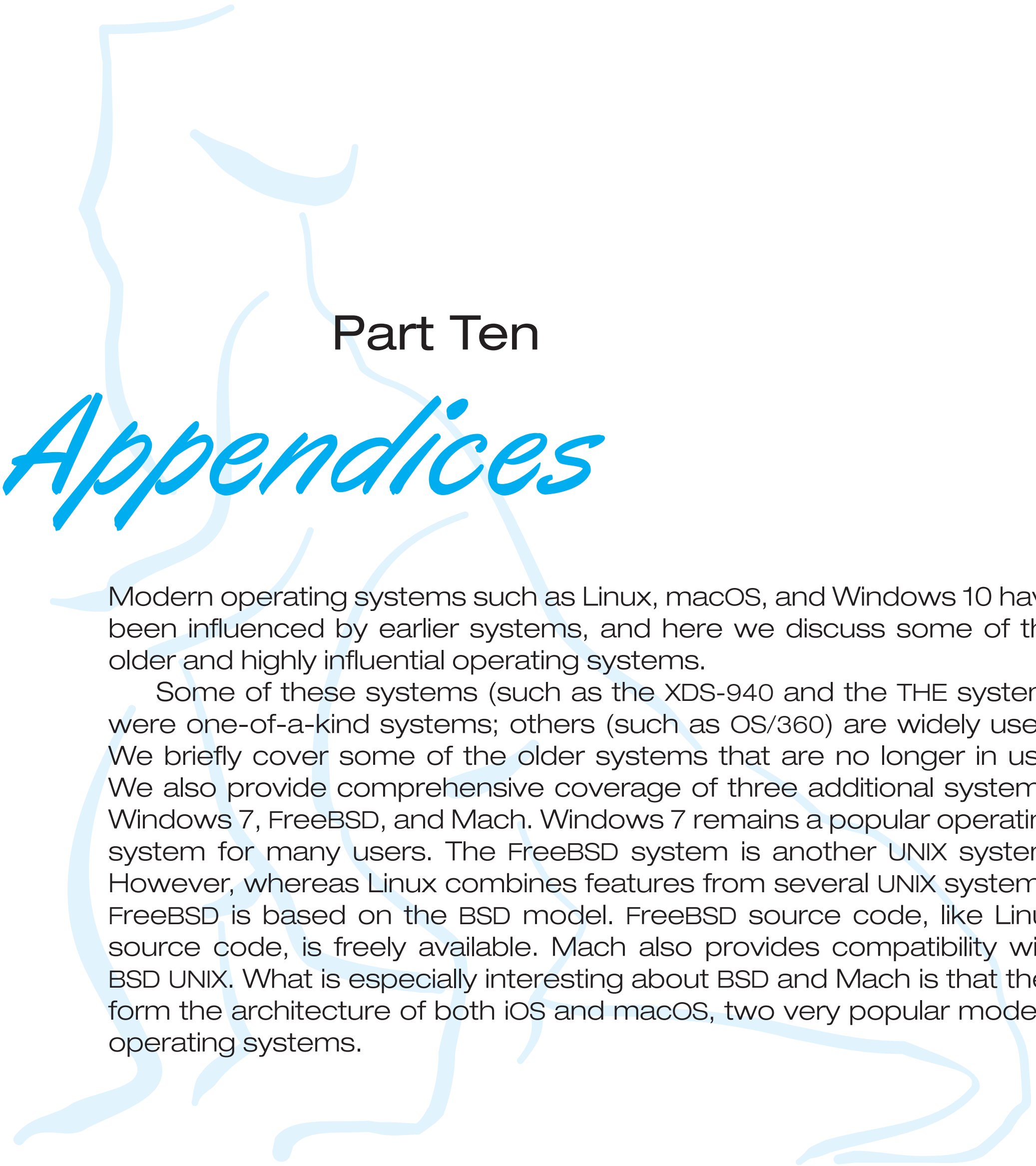
**21.25** What is the performance trade-off of allowing KTs and UTs to execute on different processors?

**21.26** Why does the self-map occupy large amounts of virtual address space but no additional virtual memory?

**21.27** How does the self-map make it easy for the VM manager to move the page-table pages to and from disk? Where are the page-table pages kept on disk?

**21.28** When a Windows system hibernates, the system is powered off. Suppose you changed the CPU or the amount of RAM on a hibernating system. Do you think that would work? Why or why not?

**21.29** Give an example showing how the use of a suspend count is helpful in suspending and resuming threads in Windows.



# Part Ten

# Appendices

Modern operating systems such as Linux, macOS, and Windows 10 have been influenced by earlier systems, and here we discuss some of the older and highly influential operating systems.

Some of these systems (such as the XDS-940 and the THE system) were one-of-a-kind systems; others (such as OS/360) are widely used. We briefly cover some of the older systems that are no longer in use. We also provide comprehensive coverage of three additional systems: Windows 7, FreeBSD, and Mach. Windows 7 remains a popular operating system for many users. The FreeBSD system is another UNIX system. However, whereas Linux combines features from several UNIX systems, FreeBSD is based on the BSD model. FreeBSD source code, like Linux source code, is freely available. Mach also provides compatibility with BSD UNIX. What is especially interesting about BSD and Mach is that they form the architecture of both iOS and macOS, two very popular modern operating systems.

# Influential Operating Systems

## Appendix A

Now that you understand the fundamental concepts of operating systems (CPU scheduling, memory management, processes, and so on), we are in a position to examine how these concepts have been applied in several older and highly influential operating systems. Some of them (such as the XDS-940 and the THE system) were one-of-a-kind systems; others (such as OS/360) are widely used. The order of presentation highlights the similarities and differences of the systems; it is not strictly chronological or ordered by importance. The serious student of operating systems should be familiar with all these systems.

In the bibliographical notes at the end of the chapter, we include references to further reading about these early systems. The papers, written by the designers of the systems, are important both for their technical content and for their style and flavor.

### CHAPTER OBJECTIVES

- Explain how operating-system features migrate over time from large computer systems to smaller ones.
- Discuss the features of several historically important operating systems.

## A.1 Feature Migration

One reason to study early architectures and operating systems is that a feature that once ran only on huge systems may eventually make its way into very small systems. Indeed, an examination of operating systems for mainframes and microcomputers shows that many features once available only on mainframes have been adopted for microcomputers. The same operating-system concepts are thus appropriate for various classes of computers: mainframes, minicomputers, microcomputers, and handhelds. To understand modern operating systems, then, you need to recognize the theme of feature migration and the long history of many operating-system features, as shown in Figure A.1.

A good example of feature migration started with the Multiplexed Information and Computing Services (MULTICS) operating system. MULTICS was devel-

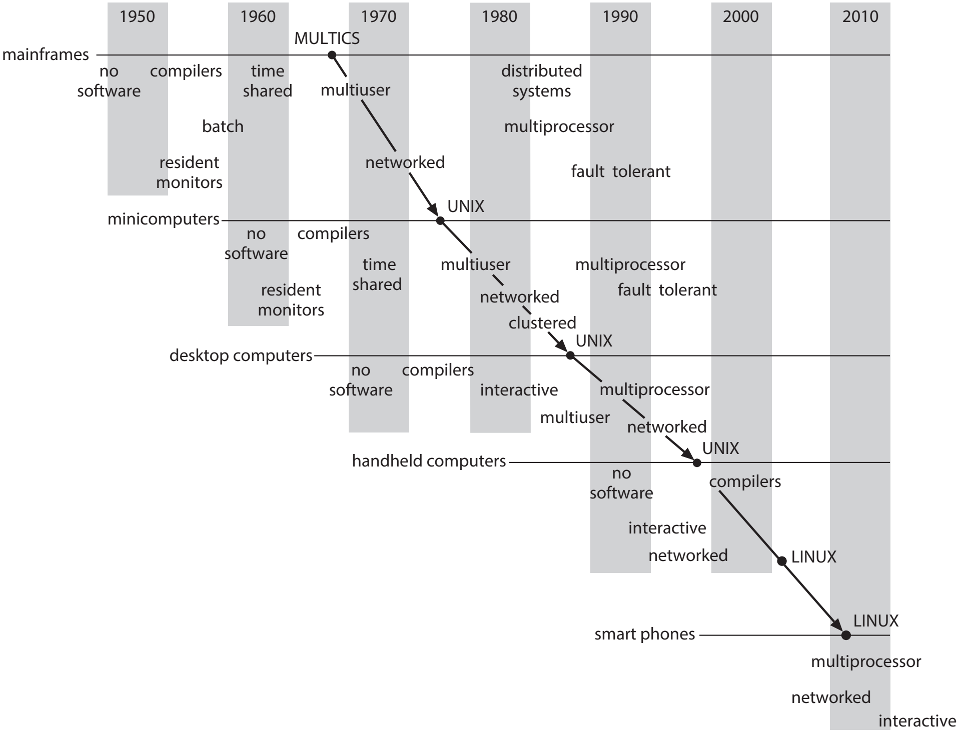

1950
1960
1970
1980
1990
2000
2010
mainframes
MULTICS
no software
compilers
time shared
multiuser
batch
resident monitors
networked
distributed systems
multiprocessor
fault tolerant
UNIX
minicomputers
no software
compilers
time shared
resident monitors
multiuser
networked
clustered
multiprocessor
fault tolerant
UNIX
desktop computers
no software
compilers
interactive
multiuser
multiprocessor
networked
UNIX
handheld computers
no software
compilers
interactive
networked
LINUX
smart phones
LINUX
multiprocessor
networked
interactive

**Figure A.1** Migration of operating-system concepts and features.

oped from 1965 to 1970 at the Massachusetts Institute of Technology (MIT) as a computing **utility**. It ran on a large, complex mainframe computer (the GE-645). Many of the ideas that were developed for MULTICS were subsequently used at Bell Laboratories (one of the original partners in the development of MULTICS) in the design of UNIX. The UNIX operating system was designed around 1970 for a PDP-11 minicomputer. Around 1980, the features of UNIX became the basis for UNIX-like operating systems on microcomputers, and these features are included in several more recent operating systems for microcomputers, such as Microsoft Windows, Windows XP, and the macOS operating system. Linux includes some of these same features, and they can now be found on PDAs.

## A.2 Early Systems

We turn our attention now to a historical overview of early computer systems. We should note that the history of computing starts far before "computers" with looms and calculators. We begin our discussion, however, with the computers of the twentieth century.

Before the 1940s, computing devices were designed and implemented to perform specific, fixed tasks. Modifying one of those tasks required a great deal of effort and manual labor. All that changed in the 1940s when Alan Turing and John von Neumann (and colleagues), both separately and together, worked on the idea of a more general-purpose **stored program** computer. Such a machine has both a program store and a data store, where the program store provides instructions about what to do to the data.

This fundamental computer concept quickly generated a number of general-purpose computers, but much of the history of these machines is blurred by time and the secrecy of their development during World War II. It is likely that the first working stored-program general-purpose computer was the Manchester Mark 1, which ran successfully in 1949. The first commercial computer—the Ferranti Mark 1, which went on sale in 1951—was its offspring.

Early computers were physically enormous machines run from consoles. The programmer, who was also the operator of the computer system, would write a program and then would operate it directly from the operator's console. First, the program would be loaded manually into memory from the front panel switches (one instruction at a time), from paper tape, or from punched cards. Then the appropriate buttons would be pushed to set the starting address and to start the execution of the program. As the program ran, the programmer/operator could monitor its execution by the display lights on the console. If errors were discovered, the programmer could halt the program, examine the contents of memory and registers, and debug the program directly from the console. Output was printed or was punched onto paper tape or cards for later printing.

### A.2.1 Dedicated Computer Systems

As time went on, additional software and hardware were developed. Card readers, line printers, and magnetic tape became commonplace. Assemblers, loaders, and linkers were designed to ease the programming task. Libraries of common functions were created. Common functions could then be copied into a new program without having to be written again, providing software reusability.

The routines that performed I/O were especially important. Each new I/O device had its own characteristics, requiring careful programming. A special subroutine called a device driver—was written for each I/O device. A device driver knows how the buffers, flags, registers, control bits, and status bits for a particular device should be used. Each type of device has its own driver. A simple task, such as reading a character from a paper-tape reader, might involve complex sequences of device-specific operations. Rather than writing the necessary code every time, the device driver was simply used from the library.

Later, compilers for FORTRAN, COBOL, and other languages appeared, making the programming task much easier but the operation of the computer more complex. To prepare a FORTRAN program for execution, for example, the programmer would first need to load the FORTRAN compiler into the computer. The compiler was normally kept on magnetic tape, so the proper tape would need to be mounted on a tape drive. The program would be read through the card reader and written onto another tape. The FORTRAN compiler produced assembly-language output, which then had to be assembled. This procedure required mounting another tape with the assembler. The output of the assembler would need to be linked to supporting library routines. Finally, the binary object form of the program would be ready to execute. It could be loaded into memory and debugged from the console, as before.

A significant amount of **setup time** could be involved in the running of a job. Each job consisted of many separate steps:

1. Loading the FORTRAN compiler tape
2. Running the compiler
3. Unloading the compiler tape
4. Loading the assembler tape
5. Running the assembler
6. Unloading the assembler tape
7. Loading the object program
8. Running the object program

If an error occurred during any step, the programmer/operator might have to start over at the beginning. Each job step might involve the loading and unloading of magnetic tapes, paper tapes, and punch cards.

The job setup time was a real problem. While tapes were being mounted or the programmer was operating the console, the CPU sat idle. Remember that, in the early days, few computers were available, and they were expensive. A computer might have cost millions of dollars, not including the operational costs of power, cooling, programmers, and so on. Thus, computer time was extremely valuable, and owners wanted their computers to be used as much as possible. They needed high **utilization** to get as much as they could from their investments.

### A.2.2 Shared Computer Systems

The solution was twofold. First, a professional computer operator was hired. The programmer no longer operated the machine. As soon as one job was finished, the operator could start the next. Since the operator had more experience with mounting tapes than a programmer, setup time was reduced. The programmer provided whatever cards or tapes were needed, as well as a short description of how the job was to be run. Of course, the operator could not debug an incorrect program at the console, since the operator would not understand the program. Therefore, in the case of program error, a dump of memory and registers was taken, and the programmer had to debug from the dump. Dumping the memory and registers allowed the operator to continue immediately with the next job but left the programmer with the more difficult debugging problem.

Second, jobs with similar needs were batched together and run through the computer as a group to reduce setup time. For instance, suppose the operator received one FORTRAN job, one COBOL job, and another FORTRAN job. If she ran them in that order, she would have to set up for FORTRAN (load the compiler tapes and so on), then set up for COBOL, and then set up for FORTRAN again. If she ran the two FORTRAN programs as a batch, however, she could setup only once for FORTRAN, saving operator time.

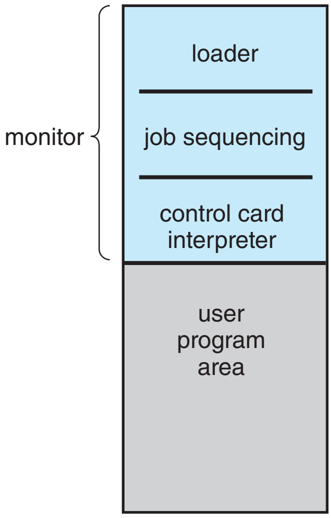


**Figure A.2** Memory layout for a resident monitor.

But there were still problems. For example, when a job stopped, the operator would have to notice that it had stopped (by observing the console), determine *why* it stopped (normal or abnormal termination), dump memory and register (if necessary), load the appropriate device with the next job, and restart the computer. During this transition from one job to the next, the CPU sat idle.

To overcome this idle time, people developed **automatic job sequencing**. With this technique, the first rudimentary operating systems were created. A small program, called a **resident monitor**, was created to transfer control automatically from one job to the next (Figure A.2). The resident monitor is always in memory (or ***resident***).

When the computer was turned on, the resident monitor was invoked, and it would transfer control to a program. When the program terminated, it would return control to the resident monitor, which would then go on to the next program. Thus, the resident monitor would automatically sequence from one program to another and from one job to another.

But how would the resident monitor know which program to execute? Previously, the operator had been given a short description of what programs were to be run on what data. **Control cards** were introduced to provide this information directly to the monitor. The idea is simple. In addition to the program or data for a job, the programmer supplied control cards, which contained directives to the resident monitor indicating what program to run. For example, a normal user program might require one of three programs to run: the FORTRAN compiler (FTN), the assembler (ASM), or the user's program (RUN). We could use a separate control card for each of these:

$FTN—Execute the FORTRAN compiler.
$ASM—Execute the assembler.
$RUN—Execute the user program.

These cards tell the resident monitor which program to run.

We can use two additional control cards to define the boundaries of each job:

$JOB—First card of a job
$END—Final card of a job

These two cards might be useful in accounting for the machine resources used by the programmer. Parameters can be used to define the job name, account number to be charged, and so on. Other control cards can be defined for other functions, such as asking the operator to load or unload a tape.

One problem with control cards is how to distinguish them from data or program cards. The usual solution is to identify them by a special character or pattern on the card. Several systems used the dollar-sign character ($) in the first column to identify a control card. Others used a different code. IBM's Job Control Language (JCL) used slash marks (//) in the first two columns. Figure A.3 shows a sample card-deck setup for a simple batch system.

A resident monitor thus has several identifiable parts:

- The **control-card interpreter** is responsible for reading and carrying out the instructions on the cards at the point of execution.
- The **loader** is invoked by the control-card interpreter to load system programs and application programs into memory at intervals.
- The **device drivers** are used by both the control-card interpreter and the loader for the system's I/O devices. Often, the system and application programs are linked to these same device drivers, providing continuity in their operation, as well as saving memory space and programming time.

These batch systems work fairly well. The resident monitor provides automatic job sequencing as indicated by the control cards. When a control card indicates that a program is to be run, the monitor loads the program into memory and transfers control to it. When the program completes, it transfers control

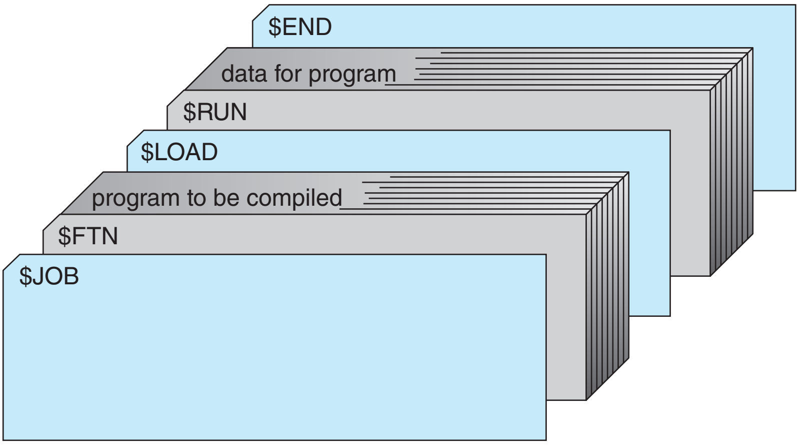


**Figure A.3** Card deck for a simple batch system.

back to the monitor, which reads the next control card, loads the appropriate program, and so on. This cycle is repeated until all control cards are interpreted for the job. Then the monitor automatically continues with the next job.

The switch to batch systems with automatic job sequencing was made to improve performance. The problem, quite simply, is that humans are considerably slower than computers. Consequently, it is desirable to replace human operation with operating-system software. Automatic job sequencing eliminates the need for human setup time and job sequencing.

Even with this arrangement, however, the CPU is often idle. The problem is the speed of the mechanical I/O devices, which are intrinsically slower than electronic devices. Even a slow CPU works in the microsecond range, with thousands of instructions executed per second. A fast card reader, in contrast, might read 1,200 cards per minute (or 20 cards per second). Thus, the difference in speed between the CPU and its I/O devices may be three orders of magnitude or more. Over time, of course, improvements in technology resulted in faster I/O devices. Unfortunately, CPU speeds increased even faster, so that the problem was not only unresolved but also exacerbated.

### A.2.3 Overlapped I/O

One common solution to the I/O problem was to replace slow card readers (input devices) and line printers (output devices) with magnetic-tape units. Most computer systems in the late 1950s and early 1960s were batch systems reading from card readers and writing to line printers or card punches. The CPU did not read directly from cards, however; instead, the cards were first copied onto a magnetic tape via a separate device. When the tape was sufficiently full, it was taken down and carried over to the computer. When a card was needed for input to a program, the equivalent record was read from the tape. Similarly, output was written to the tape, and the contents of the tape were printed later. The card readers and line printers were operated ***off-line***, rather than by the main computer (Figure A.4).

An obvious advantage of off-line operation was that the main computer was no longer constrained by the speed of the card readers and line printers but was limited only by the speed of the much faster magnetic tape units. The technique of using magnetic tape for all I/O could be applied with any

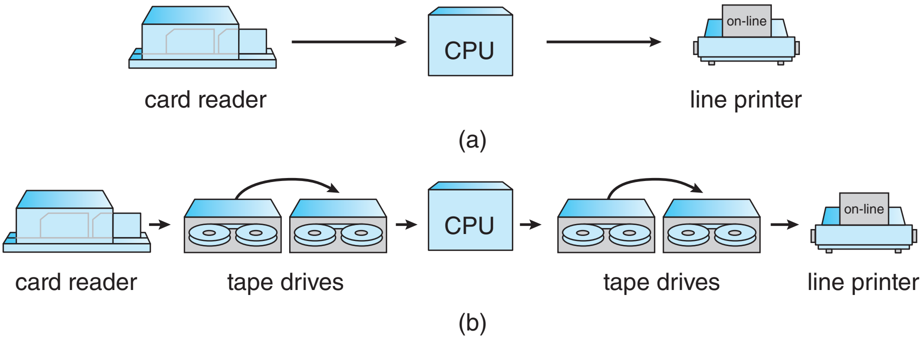


**Figure A.4** Operation of I/O devices (a) on-line and (b) off-line.

similar equipment (such as card readers, card punches, plotters, paper tape, and printers).

The real gain in off-line operation comes from the possibility of using multiple reader-to-tape and tape-to-printer systems for one CPU. If the CPU can process input twice as fast as the reader can read cards, then two readers working simultaneously can produce enough tape to keep the CPU busy. There is a disadvantage, too, however—a longer delay in getting a particular job run. The job must first be read onto tape. Then it must wait until enough additional jobs are read onto the tape to "fill" it. The tape must then be rewound, unloaded, hand-carried to the CPU, and mounted on a free tape drive. This process is not unreasonable for batch systems, of course. Many similar jobs can be batched onto a tape before it is taken to the computer.

Although off-line preparation of jobs continued for some time, it was quickly replaced in most systems. Disk systems became widely available and greatly improved on off-line operation. One problem with tape systems was that the card reader could not write onto one end of the tape while the CPU read from the other. The entire tape had to be written before it was rewound and read, because tapes are by nature **sequential-access devices**. Disk systems eliminated this problem by being **random-access devices**. Because the head is moved from one area of the disk to another, it can switch rapidly from the area on the disk being used by the card reader to store new cards to the position needed by the CPU to read the "next" card.

In a disk system, cards are read directly from the card reader onto the disk. The location of card images is recorded in a table kept by the operating system. When a job is executed, the operating system satisfies its requests for card-reader input by reading from the disk. Similarly, when the job requests the printer to output a line, that line is copied into a system buffer and is written to the disk. When the job is completed, the output is actually printed. This form of processing is called **spooling** (Figure A.5); the name is an acronym for **s**imultaneous **p**eripheral **o**peration **o**n-**l**ine. Spooling, in essence, uses the disk as a huge buffer for reading as far ahead as possible on input devices and for storing output files until the output devices are able to accept them.

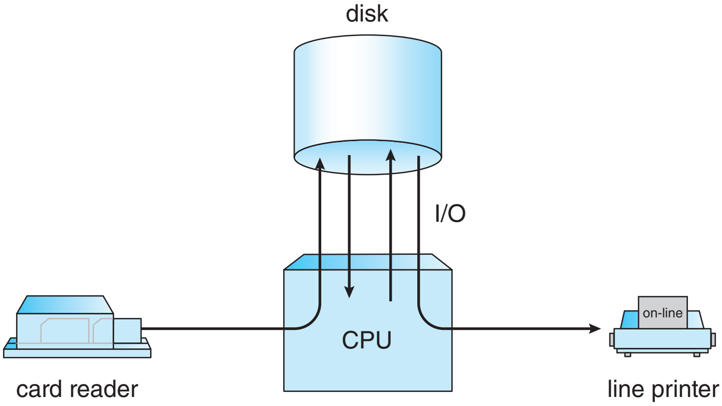


**Figure A.5** Spooling.

Spooling is also used for processing data at remote sites. The CPU sends the data via communication paths to a remote printer (or accepts an entire input job from a remote card reader). The remote processing is done at its own speed, with no CPU intervention. The CPU just needs to be notified when the processing is completed, so that it can spool the next batch of data.

Spooling overlaps the I/O of one job with the computation of other jobs. Even in a simple system, the spooler may be reading the input of one job while printing the output of a different job. During this time, still another job (or other jobs) may be executed, reading its “cards” from disk and “printing” its output lines onto the disk.

Spooling has a direct beneficial effect on the performance of the system. For the cost of some disk space and a few tables, the computation of one job and the I/O of other jobs can take place at the same time. Thus, spooling can keep both the CPU and the I/O devices working at much higher rates. Spooling leads naturally to multiprogramming, which is the foundation of all modern operating systems.

## A.3 Atlas

The Atlas operating system was designed at the University of Manchester in England in the late 1950s and early 1960s. Many of its basic features that were novel at the time have become standard parts of modern operating systems. Device drivers were a major part of the system. In addition, system calls were added by a set of special instructions called ***extra codes***.

Atlas was a batch operating system with spooling. Spooling allowed the system to schedule jobs according to the availability of peripheral devices, such as magnetic tape units, paper tape readers, paper tape punches, line printers, card readers, and card punches.

The most remarkable feature of Atlas, however, was its memory management. **Core memory** was new and expensive at the time. Many computers, like the IBM 650, used a drum for primary memory. The Atlas system used a drum for its main memory, but it had a small amount of core memory that was used as a cache for the drum. Demand paging was used to transfer information between core memory and the drum automatically.

The Atlas system used a British computer with 48-bit words. Addresses were 24 bits but were encoded in decimal, which allowed 1 million words to be addressed. At that time, this was an extremely large address space. The physical memory for Atlas was a 98-KB-word drum and 16-KB words of core. Memory was divided into 512-word pages, providing 32 frames in physical memory. An associative memory of 32 registers implemented the mapping from a virtual address to a physical address.

If a page fault occurred, a page-replacement algorithm was invoked. One memory frame was always kept empty, so that a drum transfer could start immediately. The page-replacement algorithm attempted to predict future memory-accessing behavior based on past behavior. A reference bit for each frame was set whenever the frame was accessed. The reference bits were read into memory every 1,024 instructions, and the last 32 values of these bits were retained. This history was used to define the time since the most recent reference ($t_1$) and the interval between the last two references ($t_2$). Pages were chosen for replacement in the following order:

1. Any page with $t_1 > t_2 + 1$ is considered to be no longer in use and is replaced.
2. If $t_1 \leq t_2$ for all pages, then replace the page with the largest value for $t_2 - t_1$.

The page-replacement algorithm assumes that programs access memory in loops. If the time between the last two references is $t_2$, then another reference is expected $t_2$ time units later. If a reference does not occur ($t_1 > t_2$), it is assumed that the page is no longer being used, and the page is replaced. If all pages are still in use, then the page that will not be needed for the longest time is replaced. The time to the next reference is expected to be $t_2 - t_1$.

## A.4 XDS-940

The XDS-940 operating system was designed at the University of California at Berkeley in the early 1960s. Like the Atlas system, it used paging for memory management. Unlike the Atlas system, it was a time-shared system. The paging was used only for relocation; it was not used for demand paging. The virtual memory of any user process was made up of 16-KB words, whereas the physical memory was made up of 64-KB words. Each page was made up of 2-KB words. The page table was kept in registers. Since physical memory was larger than virtual memory, several user processes could be in memory at the same time. The number of users could be increased by page sharing when the pages contained read-only reentrant code. Processes were kept on a drum and were swapped in and out of memory as necessary.

The XDS-940 system was constructed from a modified XDS-930. The modifications were typical of the changes made to a basic computer to allow an operating system to be written properly. A user-monitor mode was added. Certain instructions, such as I/O and halt, were defined to be privileged. An attempt to execute a privileged instruction in user mode would trap to the operating system.

A system-call instruction was added to the user-mode instruction set. This instruction was used to create new resources, such as files, allowing the operating system to manage the physical resources. Files, for example, were allocated in 256-word blocks on the drum. A bitmap was used to manage free drum blocks. Each file had an index block with pointers to the actual data blocks. Index blocks were chained together.

The XDS-940 system also provided system calls to allow processes to create, start, suspend, and destroy subprocesses. A programmer could construct a system of processes. Separate processes could share memory for communication and synchronization. Process creation defined a tree structure, where a process is the root and its subprocesses are nodes below it in the tree. Each of the subprocesses could, in turn, create more subprocesses.

## A.5 THE

The THE operating system was designed at the Technische Hogeschool in Eindhoven in the Netherlands in the mid-1960s. It was a batch system running on a Dutch computer, the EL X8, with 32-KB of 27-bit words. The system was mainly noted for its clean design, particularly its layer structure, and its use of a set of concurrent processes employing semaphores for synchronization.

Unlike the processes in the XDS-940 system, the set of processes in the THE system was static. The operating system itself was designed as a set of cooperating processes. In addition, five user processes were created that served as the active agents to compile, execute, and print user programs. When one job was finished, the process would return to the input queue to select another job.

A priority CPU-scheduling algorithm was used. The priorities were recomputed every 2 seconds and were inversely proportional to the amount of CPU time used recently (in the last 8 to 10 seconds). This scheme gave higher priority to I/O-bound processes and to new processes.

Memory management was limited by the lack of hardware support. However, since the system was limited and user programs could be written only in Algol, a software paging scheme was used. The Algol compiler automatically generated calls to system routines, which made sure the requested information was in memory, swapping if necessary. The backing store was a 512-KB-word drum. A 512-word page was used, with an LRU page-replacement strategy.

Another major concern of the THE system was deadlock control. The banker's algorithm was used to provide deadlock avoidance.

Closely related to the THE system is the Venus system. The Venus system was also a layer-structured design, using semaphores to synchronize processes. The lower levels of the design were implemented in microcode, however, providing a much faster system. Paged-segmented memory was used for memory management. In addition, the system was designed as a time-sharing system rather than a batch system.

## A.6 RC 4000

The RC 4000 system, like the THE system, was notable primarily for its design concepts. It was designed in the late 1960s for the Danish 4000 computer by Regnecentralen, particularly by Brinch-Hansen. The objective was not to design a batch system, or a time-sharing system, or any other specific system. Rather, the goal was to create an operating-system nucleus, or kernel, on which a complete operating system could be built. Thus, the system structure was layered, and only the lower levels—comprising the kernel—were provided.

The kernel supported a collection of concurrent processes. A round-robin CPU scheduler was used. Although processes could share memory, the primary communication and synchronization mechanism was the **message system** provided by the kernel. Processes could communicate with each other by exchanging fixed-sized messages of eight words in length. All messages were stored in buffers from a common buffer pool. When a message buffer was no longer required, it was returned to the common pool.

A **message queue** was associated with each process. It contained all the messages that had been sent to that process but had not yet been received. Messages were removed from the queue in FIFO order. The system supported four primitive operations, which were executed atomically:

- `send-message` (`in` *receiver*, `in` *message*, `out` *buffer*)
- `wait-message` (`out` *sender*, `out` *message*, `out` *buffer*)
- `send-answer` (`out` *result*, `in` *message*, `in` *buffer*)
- `wait-answer` (`out` *result*, `out` *message*, `in` *buffer*)

The last two operations allowed processes to exchange several messages at a time.

These primitives required that a process service its message queue in FIFO order and that it block itself while other processes were handling its messages. To remove these restrictions, the developers provided two additional communication primitives that allowed a process to wait for the arrival of the next message or to answer and service its queue in any order:

- `wait-event` (`in` *previous-buffer*, `out` *next-buffer*, `out` *result*)
- `get-event` (`out` *buffer*)

I/O devices were also treated as processes. The device drivers were code that converted the device interrupts and registers into messages. Thus, a process would write to a terminal by sending that terminal a message. The device driver would receive the message and output the character to the terminal. An input character would interrupt the system and transfer to a device driver. The device driver would create a message from the input character and send it to a waiting process.

## A.7 CTSS

The Compatible Time-Sharing System (CTSS) was designed at MIT as an experimental time-sharing system and first appeared in 1961. It was implemented on an IBM 7090 and eventually supported up to 32 interactive users. The users were provided with a set of interactive commands that allowed them to manipulate files and to compile and run programs through a terminal.

The 7090 had a 32-KB memory made up of 36-bit words. The monitor used 5-KB words, leaving 27 KB for the users. User memory images were swapped between memory and a fast drum. CPU scheduling employed a multilevel-feedback-queue algorithm. The time quantum for level $i$ was $2 * i$ time units. If a program did not finish its CPU burst in one time quantum, it was moved down to the next level of the queue, giving it twice as much time. The program at the highest level (with the shortest quantum) was run first. The initial level of a program was determined by its size, so that the time quantum was at least as long as the swap time.

CTSS was extremely successful and was in use as late as 1972. Although it was limited, it succeeded in demonstrating that time sharing was a convenient and practical mode of computing. One result of CTSS was increased development of time-sharing systems. Another result was the development of MULTICS.

## A.8 MULTICS

The MULTICS operating system was designed from 1965 to 1970 at MIT as a natural extension of CTSS. CTSS and other early time-sharing systems were so successful that they created an immediate desire to proceed quickly to bigger and better systems. As larger computers became available, the designers of CTSS set out to create a time-sharing utility. Computing service would be provided like electrical power. Large computer systems would be connected by telephone wires to terminals in offices and homes throughout a city. The operating system would be a time-shared system running continuously with a vast file system of shared programs and data.

MULTICS was designed by a team from MIT, GE (which later sold its computer department to Honeywell), and Bell Laboratories (which dropped out of the project in 1969). The basic GE 635 computer was modified to a new computer system called the GE 645, mainly by the addition of paged-segmentation memory hardware.

In MULTICS, a virtual address was composed of an 18-bit segment number and a 16-bit word offset. The segments were then paged in 1-KB-word pages. The second-chance page-replacement algorithm was used.

The segmented virtual address space was merged into the file system; each segment was a file. Segments were addressed by the name of the file. The file system itself was a multilevel tree structure, allowing users to create their own subdirectory structures.

Like CTSS, MULTICS used a multilevel feedback queue for CPU scheduling. Protection was accomplished through an access list associated with each file and a set of protection rings for executing processes. The system, which was written almost entirely in PL/1, comprised about 300,000 lines of code. It was extended to a multiprocessor system, allowing a CPU to be taken out of service for maintenance while the system continued running.

## A.9 IBM OS/360

The longest line of operating-system development is undoubtedly that of IBM computers. The early IBM computers, such as the IBM 7090 and the IBM 7094, are prime examples of the development of common I/O subroutines, followed by development of a resident monitor, privileged instructions, memory protection, and simple batch processing. These systems were developed separately, often at independent sites. As a result, IBM was faced with many different computers, with different languages and different system software.

The IBM/360—which first appeared in the mid 1960s—was designed to alter this situation. The IBM/360 was designed as a family of computers spanning the complete range from small business machines to large scientific machines. Only one set of software would be needed for these systems, which all used the same operating system: OS/360. This arrangement was intended to reduce maintenance problems for IBM and to allow users to move programs and applications freely from one IBM system to another.

Unfortunately, OS/360 tried to be all things to all people. As a result, it did none of its tasks especially well. The file system included a type field that defined the type of each file, and different file types were defined for fixed-length and variable-length records and for blocked and unblocked files. Contiguous allocation was used, so the user had to guess the size of each output file. The Job Control Language (JCL) added parameters for every possible option, making it incomprehensible to the average user.

The memory-management routines were hampered by the architecture. Although a base-register addressing mode was used, the program could access and modify the base register, so that absolute addresses were generated by the CPU. This arrangement prevented dynamic relocation; the program was bound to physical memory at load time. Two separate versions of the operating system were produced: OS/MFT used fixed regions and OS/MVT used variable regions.

The system was written in assembly language by thousands of programmers, resulting in millions of lines of code. The operating system itself required large amounts of memory for its code and tables. Operating-system overhead often consumed one-half of the total CPU cycles. Over the years, new versions were released to add new features and to fix errors. However, fixing one error often caused another in some remote part of the system, so that the number of known errors in the system remained fairly constant.

Virtual memory was added to OS/360 with the change to the IBM/370 architecture. The underlying hardware provided a segmented-paged virtual memory. New versions of OS used this hardware in different ways. OS/VS1 created one large virtual address space and ran OS/MFT in that virtual memory. Thus, the operating system itself was paged, as well as user programs. OS/VS2 Release 1 ran OS/MVT in virtual memory. Finally, OS/VS2 Release 2, which is now called MVS, provided each user with his own virtual memory.

MVS is still basically a batch operating system. The CTSS system was run on an IBM 7094, but the developers at MIT decided that the address space of the 360, IBM's successor to the 7094, was too small for MULTICS, so they switched vendors. IBM then decided to create its own time-sharing system, TSS/360. Like MULTICS, TSS/360 was supposed to be a large, time-shared utility. The basic 360 architecture was modified in the model 67 to provide virtual memory. Several sites purchased the 360/67 in anticipation of TSS/360.

TSS/360 was delayed, however, so other time-sharing systems were developed as temporary systems until TSS/360 was available. A time-sharing option (TSO) was added to OS/360. IBM's Cambridge Scientific Center developed CMS as a single-user system and CP/67 to provide a virtual machine to run it on.

When TSS/360 was eventually delivered, it was a failure. It was too large and too slow. As a result, no site would switch from its temporary system to TSS/360. Today, time sharing on IBM systems is largely provided either by TSO under MVS or by CMS under CP/67 (renamed VM).

Neither TSS/360 nor MULTICS achieved commercial success. What went wrong? Part of the problem was that these advanced systems were too large and too complex to be understood. Another problem was the assumption that computing power would be available from a large, remote source. Minicomputers came along and decreased the need for large monolithic systems. They were followed by workstations and then personal computers, which put computing power closer and closer to the end users.

## A.10 TOPS-20

DEC created many influential computer systems during its history. Probably the most famous operating system associated with DEC is VMS, a popular business-oriented system that is still in use today as OpenVMS, a product of Hewlett-Packard. But perhaps the most influential of DEC's operating systems was TOPS-20.

TOPS-20 started life as a research project at Bolt, Beranek, and Newman (BBN) around 1970. BBN took the business-oriented DEC PDP-10 computer running TOPS-10, added a hardware memory-paging system to implement virtual memory, and wrote a new operating system for that computer to take advantage of the new hardware features. The result was TENEX, a general-purpose time-sharing system. DEC then purchased the rights to TENEX and created a new computer with a built-in hardware pager. The resulting system was the DECSYSTEM-20 and the TOPS-20 operating system.

TOPS-20 had an advanced command-line interpreter that provided help as needed to users. That, in combination with the power of the computer and its reasonable price, made the DECSYSTEM-20 the most popular time-sharing system of its time. In 1984, DEC stopped work on its line of 36-bit PDP-10 computers to concentrate on 32-bit VAX systems running VMS.

## A.11 CP/M and MS/DOS

Early hobbyist computers were typically built from kits and ran a single program at a time. The systems evolved into more advanced systems as computer components improved. An early "standard" operating system for these computers of the 1970s was **CP/M**, short for Control Program/Monitor, written by Gary Kindall of Digital Research, Inc. CP/M ran primarily on the first "personal computer" CPU, the 8-bit Intel 8080. CP/M originally supported only 64 KB of memory and ran only one program at a time. Of course, it was text-based, with a command interpreter. The command interpreter resembled those in other operating systems of the time, such as the TOPS-10 from DEC.

When IBM entered the personal computer business, it decided to have Bill Gates and company write a new operating system for its 16-bit CPU of choice—the Intel 8086. This operating system, **MS-DOS**, was similar to CP/M but had a richer set of built-in commands, again mostly modeled after TOPS-10. MS-DOS became the most popular personal-computer operating system of its time, starting in 1981 and continuing development until 2000. It supported 640 KB of memory, with the ability to address "extended" and "expanded" memory to get somewhat beyond that limit. It lacked fundamental current operating-system features, however, especially protected memory.

## A.12 Macintosh Operating System and Windows

With the advent of 16-bit CPUs, operating systems for personal computers could become more advanced, feature rich, and usable. The **Apple Macintosh** computer was arguably the first computer with a GUI designed for home users. It was certainly the most successful for a while, starting at its launch in 1984. It used a mouse for screen pointing and selecting and came with many utility programs that took advantage of the new user interface. Hard-disk drives were relatively expensive in 1984, so it came only with a 400-KB-capacity floppy drive by default.

The original Mac OS ran only on Apple computers and slowly was eclipsed by Microsoft Windows (starting with Version 1.0 in 1985), which was licensed to run on many different computers from a multitude of companies. As microprocessor CPUs evolved to 32-bit chips with advanced features, such as protected memory and context switching, these operating systems added features that had previously been found only on mainframes and minicomputers. Over time, personal computers became as powerful as those systems and more useful for many purposes. Minicomputers died out, replaced by general- and special-purpose "servers." Although personal computers continue to increase in capacity and performance, servers tend to stay ahead of them in amount of memory, disk space, and number and speed of available CPUs. Today, servers typically run in data centers or machine rooms, while personal computers sit on or next to desks and talk to each other and servers across a network.

The desktop rivalry between Apple and Microsoft continues today, with new versions of Windows and Mac OS trying to outdo each other in features, usability, and application functionality. Other operating systems, such as AmigaOS and OS/2, have appeared over time but have not been long-term competitors to the two leading desktop operating systems. Meanwhile, Linux in its many forms continues to gain in popularity among more technical users —and even with nontechnical users on systems like the **One Laptop per Child** (**OLPC**) children's connected computer network (http://laptop.org/).

## A.13 Mach

The Mach operating system traces its ancestry to the Accent operating system developed at Carnegie Mellon University (CMU). Mach's communication system and philosophy are derived from Accent, but many other significant portions of the system (for example, the virtual memory system and task and thread management) were developed from scratch.

Work on Mach began in the mid 1980. The operating system was designed with the following three critical goals in mind:

1. Emulate 4.3 BSD UNIX so that the executable files from a UNIX system can run correctly under Mach.
2. Be a modern operating system that supports many memory models, as well as parallel and distributed computing.
3. Have a kernel that is simpler and easier to modify than 4.3 BSD.

Mach's development followed an evolutionary path from BSD UNIX systems. Mach code was initially developed inside the 4.2 BSD kernel, with BSD kernel components replaced by Mach components as the Mach components were completed. The BSD components were updated to 4.3 BSD when that became available. By 1986, the virtual memory and communication subsystems were running on the DEC VAX computer family, including multiprocessor versions of the VAX. Versions for the IBM RT/PC and for SUN 3 workstations followed shortly. Then, 1987 saw the completion of the Encore Multimax and Sequent Balance multiprocessor versions, including task and thread support, as well as the first official releases of the system, Release 0 and Release 1.

Through Release 2, Mach provided compatibility with the corresponding BSD systems by including much of BSD's code in the kernel. The new features and capabilities of Mach made the kernels in these releases larger than the corresponding BSD kernels. Mach 3 moved the BSD code outside the kernel, leaving a much smaller microkernel. This system implements only basic Mach features in the kernel; all UNIX-specific code has been evicted to run in user-mode servers. Excluding UNIX-specific code from the kernel allows the replacement of BSD with another operating system or the simultaneous execution of multiple operating-system interfaces on top of the microkernel. In addition to BSD, user-mode implementations have been developed for DOS, the Macintosh operating system, and OSF/1. This approach has similarities to the virtual machine concept, but here the virtual machine is defined by software (the Mach kernel interface), rather than by hardware. With Release 3.0, Mach became available on a wide variety of systems, including single-processor SUN, Intel, IBM, and DEC machines and multiprocessor DEC, Sequent, and Encore systems.

Mach was propelled to the forefront of industry attention when the Open Software Foundation (OSF) announced in 1989 that it would use Mach 2.5 as the basis for its new operating system, OSF/1. (Mach 2.5 was also the basis for the operating system on the NeXT workstation, the brainchild of Steve Jobs of Apple Computer fame.) The initial release of OSF/1 occurred a year later, and this system competed with UNIX System V, Release 4, the operating system of choice at that time among UNIX International (UI) members. OSF members included key technological companies such as IBM, DEC, and HP. OSF has since changed its direction, and only DEC UNIX is based on the Mach kernel.

Unlike UNIX, which was developed without regard for multiprocessing, Mach incorporates multiprocessing support throughout. This support is also exceedingly flexible, ranging from shared-memory systems to systems with no memory shared between processors. Mach uses lightweight processes, in the form of multiple threads of execution within one task (or address space), to support multiprocessing and parallel computation. Its extensive use of messages as the only communication method ensures that protection mechanisms are complete and efficient. By integrating messages with the virtual memory system, Mach also ensures that messages can be handled efficiently. Finally, by having the virtual memory system use messages to communicate with the daemons managing the backing store, Mach provides great flexibility in the design and implementation of these memory-object-managing tasks. By providing low-level, or primitive, system calls from which more complex functions can be built, Mach reduces the size of the kernel while permitting operating-system emulation at the user level, much like IBM's virtual machine systems.

Today, the only remaining pure Mach implementation is in GNU HURD, a little-used operating system. Mach still lives on, however, in XNU—the kernel driving macOSand the iOSvariants. XNU—whose codebase Apple obtained with the acquisition of NeXT and its NeXTSTEP operating system—is a Mach core with a top layer of BSD APIs. Apple continues to support and maintain the Mach APIs (still accessible through specialized system calls known as traps, and via Mach Messages), and the kernel continues evolving with new features to this day.

Some previous editions of ***Operating System Concepts*** included an entire chapter on Mach. This chapter, as it appeared in the fourth edition, is available on the web (http://www.os-book.com).

## A.14 Capability-based Systems—Hydra and CAP

In this section, we survey two capability-based protection systems. These systems differ in their complexity and in the types of policies that can be implemented on them. Neither system is widely used, but both provide interesting proving grounds for protection theories.

### A.14.1 Hydra

Hydra is a capability-based protection system that provides considerable flexibility. The system implements a fixed set of possible access rights, including such basic forms of access as the right to read, write, or execute a memory segment. In addition, a user (of the protection system) can declare other rights. The interpretation of user-defined rights is performed solely by the user's program, but the system provides access protection for the use of these rights, as well as for the use of system-defined rights. These facilities constitute a significant development in protection technology.

Operations on objects are defined procedurally. The procedures that implement such operations are themselves a form of object, and they are accessed indirectly by capabilities. The names of user-defined procedures must be identified to the protection system if it is to deal with objects of the user-defined type. When the definition of an object is made known to Hydra, the names of operations on the type become **auxiliary rights**. Auxiliary rights can be described in a capability for an instance of the type. For a process to perform an operation on a typed object, the capability it holds for that object must contain the name of the operation being invoked among its auxiliary rights. This restriction enables discrimination of access rights to be made on an instance-by-instance and process-by-process basis.

Hydra also provides **rights amplificatio** . This scheme allows a procedure to be certified as ***trustworthy*** to act on a formal parameter of a specified type on behalf of any process that holds a right to execute the procedure. The rights held by a trustworthy procedure are independent of, and may exceed, the rights held by the calling process. However, such a procedure must not be regarded as universally trustworthy (the procedure is not allowed to act on other types, for instance), and the trustworthiness must not be extended to any other procedures or program segments that might be executed by a process.

Amplification allows implementation procedures access to the representation variables of an abstract data type. If a process holds a capability to a typed object *A*, for instance, this capability may include an auxiliary right to invoke some operation *P* but does not include any of the so-called kernel rights, such as read, write, or execute, on the segment that represents *A*. Such a capability gives a process a means of indirect access (through the operation *P*) to the representation of *A*, but only for specific purposes.

When a process invokes the operation *P* on an object *A*, however, the capability for access to *A* may be amplified as control passes to the code body of *P*. This amplification may be necessary to allow *P* the right to access the storage segment representing *A* so as to implement the operation that *P* defines on the abstract data type. The code body of *P* may be allowed to read or to write to the segment of *A* directly, even though the calling process cannot. On return from *P*, the capability for *A* is restored to its original, unamplified state. This case is a typical one in which the rights held by a process for access to a protected segment must change dynamically, depending on the task to be performed. The dynamic adjustment of rights is performed to guarantee consistency of a programmer-defined abstraction. Amplification of rights can be stated explicitly in the declaration of an abstract type to the Hydra operating system.

When a user passes an object as an argument to a procedure, we may need to ensure that the procedure cannot modify the object. We can implement this restriction readily by passing an access right that does not have the modification (write) right. However, if amplification may occur, the right to modify may be reinstated. Thus, the user-protection requirement can be circumvented. In general, of course, a user may trust that a procedure performs its task correctly. This assumption is not always correct, however, because of hardware or software errors. Hydra solves this problem by restricting amplifications.

The procedure-call mechanism of Hydra was designed as a direct solution to the ***problem of mutually suspicious subsystems***. This problem is defined as follows. Suppose that a program can be invoked as a service by a number of different users (for example, a sort routine, a compiler, a game). When users invoke this service program, they take the risk that the program will malfunction and will either damage the given data or retain some access right to the data to be used (without authority) later. Similarly, the service program may have some private files (for accounting purposes, for example) that should not be accessed directly by the calling user program. Hydra provides mechanisms for directly dealing with this problem.

A Hydra subsystem is built on top of its protection kernel and may require protection of its own components. A subsystem interacts with the kernel through calls on a set of kernel-defined primitives that define access rights to resources defined by the subsystem. The subsystem designer can define policies for use of these resources by user processes, but the policies are enforced by use of the standard access protection provided by the capability system.

Programmers can make direct use of the protection system after acquainting themselves with its features in the appropriate reference manual. Hydra provides a large library of system-defined procedures that can be called by user programs. Programmers can explicitly incorporate calls on these system procedures into their program code or can use a program translator that has been interfaced to Hydra.

### A.14.2 Cambridge CAP System

A different approach to capability-based protection has been taken in the design of the Cambridge CAP system. CAP's capability system is simpler and superficially less powerful than that of Hydra. However, closer examination shows that it, too, can be used to provide secure protection of user-defined objects. CAP has two kinds of capabilities. The ordinary kind is called a **data capability**. It can be used to provide access to objects, but the only rights provided are the standard read, write, and execute of the individual storage segments associated with the object. Data capabilities are interpreted by microcode in the CAP machine.

The second kind of capability is the so-called **software capability**, which is protected, but not interpreted, by the CAP microcode. It is interpreted by a ***protected*** (that is, privileged) procedure, which may be written by an application programmer as part of a subsystem. A particular kind of rights amplification is associated with a protected procedure. When executing the code body of such a procedure, a process temporarily acquires the right to read or write the contents of a software capability itself. This specific kind of rights amplification corresponds to an implementation of the `seal` and `unseal` primitives on capabilities. Of course, this privilege is still subject to type verification to ensure that only software capabilities for a specified abstract type are passed to any such procedure. Universal trust is not placed in any code other than the CAP machine's microcode. (See the bibliographical notes at the end of the chapter for references.)

The interpretation of a software capability is left completely to the subsystem, through the protected procedures it contains. This scheme allows a variety of protection policies to be implemented. Although programmers can define their own protected procedures (any of which might be incorrect), the security of the overall system cannot be compromised. The basic protection system will not allow an unverified, user-defined, protected procedure access to any storage segments (or capabilities) that do not belong to the protection environment in which it resides. The most serious consequence of an insecure protected procedure is a protection breakdown of the subsystem for which that procedure has responsibility.

The designers of the CAP system have noted that the use of software capabilities allowed them to realize considerable economies in formulating and implementing protection policies commensurate with the requirements of abstract resources. However, subsystem designers who want to make use of this facility cannot simply study a reference manual, as is the case with Hydra. Instead, they must learn the principles and techniques of protection, since the system provides them with no library of procedures.

## A.15 Other Systems

There are, of course, other operating systems, and most of them have interesting properties. The MCP operating system for the Burroughs computer family was the first to be written in a system programming language. It supported segmentation and multiple CPUs. The SCOPE operating system for the CDC 6600 was also a multi-CPU system. The coordination and synchronization of the multiple processes were surprisingly well designed.

History is littered with operating systems that suited a purpose for a time (be it a long or a short time) and then, when faded, were replaced by operating systems that had more features, supported newer hardware, were easier to use, or were better marketed. We are sure this trend will continue in the future.

## Further Reading

Looms and calculators are described in [Frah (2001)] and shown graphically in [Frauenfelder (2005)].

The Manchester Mark 1 is discussed by [Rojas and Hashagen (2000)], and its offspring, the Ferranti Mark 1, is described by [Ceruzzi (1998)].

[Kilburn et al. (1961)] and [Howarth et al. (1961)] examine the Atlas operating system.

The XDS-940 operating system is described by [Lichtenberger and Pirtle (1965)].

The THE operating system is covered by [Dijkstra (1968)] and by [McKeag and Wilson (1976)].

The Venus system is described by [Liskov (1972)].

[Brinch-Hansen (1970)] and [Brinch-Hansen (1973)] discuss the RC 4000 system.

The Compatible Time-Sharing System (CTSS) is presented by [Corbato et al. (1962)].

The MULTICS operating system is described by [Corbato and Vyssotsky (1965)] and [Organick (1972)].

[Mealy et al. (1966)] presented the IBM/360. [Lett and Konigsford (1968)] cover TSS/360.

CP/67 is described by [Meyer and Seawright (1970)] and [Parmelee et al. (1972)].

DEC VMS is discussed by [Kenah et al. (1988)], and TENEX is described by [Bobrow et al. (1972)].

A description of the Apple Macintosh appears in [Apple (1987)]. For more information on these operating systems and their history, see [Freiberger and Swaine (2000)].

The Mach operating system and its ancestor, the Accent operating system, are described by [Rashid and Robertson (1981)]. Mach's communication system is covered by [Rashid (1986)], [Tevanian et al. (1989)], and [Accetta et al. (1986)]. The Mach scheduler is described in detail by [Tevanian et al. (1987a)] and [Black (1990)]. An early version of the Mach shared-memory and memory-mapping system is presented by [Tevanian et al. (1987b)]. A good resource describing the Mach project can be found at http://www.cs.cmu.edu/afs/cs/project/mach/public/www/mach.html.

[McKeag and Wilson (1976)] discuss the MCP operating system for the Burroughs computer family as well as the SCOPE operating system for the CDC 6600.

The Hydra system was described by [Wulf et al. (1981)]. The CAP system was described by [Needham and Walker (1977)]. [Organick (1972)] discussed the MULTICS ring-protection system.

## Bibliography

**[Accetta et al. (1986)]** M. Accetta, R. Baron, W. Bolosky, D. B. Golub, R. Rashid, A. Tevanian, and M. Young, “Mach: A New Kernel Foundation for UNIX Development”, *Proceedings of the Summer USENIX Conference* (1986), pages 93–112.

**[Apple (1987)]** *Apple Technical Introduction to the Macintosh Family*. Addison-Wesley (1987).

**[Black (1990)]** D. L. Black, “Scheduling Support for Concurrency and Parallelism in the Mach Operating System”, *IEEE Computer*, Volume 23, Number 5 (1990), pages 35–43.

**[Bobrow et al. (1972)]** D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson, “TENEX, a Paged Time Sharing System for the PDP-10”, *Communications of the ACM*, Volume 15, Number 3 (1972).

**[Brinch-Hansen (1970)]** P. Brinch-Hansen, “The Nucleus of a Multiprogramming System”, *Communications of the ACM*, Volume 13, Number 4 (1970), pages 238–241 and 250.

**[Brinch-Hansen (1973)]** P. Brinch-Hansen, *Operating System Principles*, Prentice Hall (1973).

**[Ceruzzi (1998)]** P. E. Ceruzzi, *A History of Modern Computing*, MIT Press (1998).

**[Corbato and Vyssotsky (1965)]** F. J. Corbato and V. A. Vyssotsky, “Introduction and Overview of the MULTICS System”, *Proceedings of the AFIPS Fall Joint Computer Conference* (1965), pages 185–196.

**[Corbato et al. (1962)]** F. J. Corbato, M. Merwin-Daggett, and R. C. Daley, “An Experimental Time-Sharing System”, *Proceedings of the AFIPS Fall Joint Computer Conference* (1962), pages 335–344.

**[Dijkstra (1968)]** E. W. Dijkstra, “The Structure of the THE Multiprogramming System”, *Communications of the ACM*, Volume 11, Number 5 (1968), pages 341–346.

**[Frah (2001)]** G. Frah, *The Universal History of Computing*, John Wiley and Sons (2001).

**[Frauenfelder (2005)]** M. Frauenfelder, *The Computer—An Illustrated History*, Carlton Books (2005).

**[Freiberger and Swaine (2000)]** P. Freiberger and M. Swaine, *Fire in the Valley—The Making of the Personal Computer*, McGraw-Hill (2000).

**[Howarth et al. (1961)]** D. J. Howarth, R. B. Payne, and F. H. Sumner, “The Manchester University Atlas Operating System, Part II: User’s Description”, *Computer Journal*, Volume 4, Number 3 (1961), pages 226–229.

**[Kenah et al. (1988)]** L. J. Kenah, R. E. Goldenberg, and S. F. Bate, *VAX/VMS Internals and Data Structures*, Digital Press (1988).

**[Kilburn et al. (1961)]** T. Kilburn, D. J. Howarth, R. B. Payne, and F. H. Sumner, “The Manchester University Atlas Operating System, Part I: Internal Organization”, *Computer Journal*, Volume 4, Number 3 (1961), pages 222–225.

**[Lett and Konigsford (1968)]** A. L. Lett and W. L. Konigsford, “TSS/360: A Time-Shared Operating System”, *Proceedings of the AFIPS Fall Joint Computer Conference* (1968), pages 15–28.

**[Lichtenberger and Pirtle (1965)]** W. W. Lichtenberger and M. W. Pirtle, “A Facility for Experimentation in Man-Machine Interaction”, *Proceedings of the AFIPS Fall Joint Computer Conference* (1965), pages 589–598.

**[Liskov (1972)]** B. H. Liskov, “The Design of the Venus Operating System”, *Communications of the ACM*, Volume 15, Number 3 (1972), pages 144–149.

**[McKeag and Wilson (1976)]** R. M. McKeag and R. Wilson, *Studies in Operating Systems*, Academic Press (1976).

**[Mealy et al. (1966)]** G. H. Mealy, B. I. Witt, and W. A. Clark, “The Functional Structure of OS/360”, *IBM Systems Journal*, Volume 5, Number 1 (1966), pages 3–11.

**[Meyer and Seawright (1970)]** R. A. Meyer and L. H. Seawright, “A Virtual Machine Time-Sharing System”, *IBM Systems Journal*, Volume 9, Number 3 (1970), pages 199–218.

**[Needham and Walker (1977)]** R. M. Needham and R. D. H. Walker, “The Cambridge CAP Computer and Its Protection System”, *Proceedings of the Sixth Symposium on Operating System Principles* (1977), pages 1–10.

**[Organick (1972)]** E. I. Organick, *The Multics System: An Examination of Its Structure*, MIT Press (1972).

**[Parmelee et al. (1972)]** R. P. Parmelee, T. I. Peterson, C. C. Tillman, and D. Hatfield, “Virtual Storage and Virtual Machine Concepts”, *IBM Systems Journal*, Volume 11, Number 2 (1972), pages 99–130.

**[Rashid (1986)]** R. F. Rashid, “From RIG to Accent to Mach: The Evolution of a Network Operating System”, *Proceedings of the ACM/IEEE Computer Society, Fall Joint Computer Conference* (1986), pages 1128–1137.

**[Rashid and Robertson (1981)]** R. Rashid and G. Robertson, “Accent: A Communication-Oriented Network Operating System Kernel”, *Proceedings of the ACM Symposium on Operating System Principles* (1981), pages 64–75.

**[Rojas and Hashagen (2000)]** R. Rojas and U. Hashagen, *The First Computers—History and Architectures*, MIT Press (2000).

**[Tevanian et al. (1987a)]** A. Tevanian, Jr., R. F. Rashid, D. B. Golub, D. L. Black, E. Cooper, and M. W. Young, “Mach Threads and the Unix Kernel: The Battle for Control”, *Proceedings of the Summer USENIX Conference* (1987).

**[Tevanian et al. (1987b)]** A. Tevanian, Jr., R. F. Rashid, M. W. Young, D. B. Golub, M. R. Thompson, W. Bolosky, and R. Sanzi, “A UNIX Interface for Shared Memory and Memory Mapped Files Under Mach”, Technical report, Carnegie-Mellon University (1987).

**[Tevanian et al. (1989)]** A. Tevanian, Jr., and B. Smith, “Mach: The Model for Future Unix”, *Byte* (1989).

**[Wulf et al. (1981)]** W. A. Wulf, R. Levin, and S. P. Harbison, *Hydra/C.mmp: An Experimental Computer System*, McGraw-Hill (1981).

# Appendix B

# Windows 7

**Updated by Dave Probert**

The Microsoft Windows 7 operating system is a 32-/64-bit preemptive multitasking client operating system for microprocessors implementing the Intel IA-32 and AMD64 instruction set architectures (ISAs). Microsoft's corresponding server operating system, Windows Server 2008 R2, is based on the same code as Windows 7 but supports only the 64-bit AMD64 and IA64 (Itanium) ISAs. Windows 7 is the latest in a series of Microsoft operating systems based on its NT code, which replaced the earlier systems based on Windows 95/98. In this appendix, we discuss the key goals of Windows 7, the layered architecture of the system that has made it so easy to use, the file system, the networking features, and the programming interface.

## CHAPTER OBJECTIVES

- Explore the principles underlying Windows 7's design and the specific components of the system.
- Provide a detailed discussion of the Windows 7 file system.
- Illustrate the networking protocols supported in Windows 7.
- Describe the interface available in Windows 7 to system and application programmers.
- Describe the important algorithms implemented with Windows 7.

## B.1 History

In the mid-1980s, Microsoft and IBM cooperated to develop the **OS/2 operating system**, which was written in assembly language for single-processor Intel 80286 systems. In 1988, Microsoft decided to end the joint effort with IBM and develop its own "new technology" (or NT) portable operating system to support both the OS/2 and POSIX application-programming interfaces (APIs). In

October 1988, Dave Cutler, the architect of the DEC VAX/VMS operating system, was hired and given the charter of building Microsoft's new operating system.

Originally, the team planned to use the OS/2 API as NT's native environment, but during development, NT was changed to use a new 32-bit Windows API (called Win32), based on the popular 16-bit API used in Windows 3.0. The first versions of NT were Windows NT 3.1 and Windows NT 3.1 Advanced Server. (At that time, 16-bit Windows was at Version 3.1.) Windows NT Version 4.0 adopted the Windows 95 user interface and incorporated Internet web-server and web-browser software. In addition, user-interface routines and all graphics code were moved into the kernel to improve performance, with the side effect of decreased system reliability. Although previous versions of NT had been ported to other microprocessor architectures, the Windows 2000 version, released in February 2000, supported only Intel (and compatible) processors due to marketplace factors. Windows 2000 incorporated significant changes. It added Active Directory (an X.500-based directory service), better networking and laptop support, support for plug-and-play devices, a distributed file system, and support for more processors and more memory.

In October 2001, Windows XP was released as both an update to the Windows 2000 desktop operating system and a replacement for Windows 95/98. In 2002, the server edition of Windows XP became available (called Windows .Net Server). Windows XP updated the graphical user interface (GUI) with a visual design that took advantage of more recent hardware advances and many new ***ease-of-use features***. Numerous features were added to automatically repair problems in applications and the operating system itself. As a result of these changes, Windows XP provided better networking and device experience (including zero-configuration wireless, instant messaging, streaming media, and digital photography/video), dramatic performance improvements for both the desktop and large multiprocessors, and better reliability and security than earlier Windows operating systems.

The long-awaited update to Windows XP, called Windows Vista, was released in November 2006, but it was not well received. Although Windows Vista included many improvements that later showed up in Windows 7, these improvements were overshadowed by Windows Vista's perceived sluggishness and compatibility problems. Microsoft responded to criticisms of Windows Vista by improving its engineering processes and working more closely with the makers of Windows hardware and applications. The result was **Windows 7**, which was released in October 2009, along with corresponding server editions of Windows. Among the significant engineering changes is the increased use of **execution tracing** rather than counters or profiling to analyze system behavior. Tracing runs constantly in the system, watching hundreds of scenarios execute. When one of these scenarios fails, or when it succeeds but does not perform well, the traces can be analyzed to determine the cause.

Windows 7 uses a client–server architecture (like Mach) to implement two operating-system personalities, Win32 and POSIX, with user-level processes called subsystems. (At one time, Windows also supported an OS/2 subsystem, but it was removed in Windows XP due to the demise of OS/2.) The subsystem architecture allows enhancements to be made to one operating-system personality without affecting the application compatibility of the other. Although the POSIX subsystem continues to be available for Windows 7, the Win32 API has become very popular, and the POSIX APIs are used by only a few sites. The subsystem approach continues to be interesting to study from an operating-system perspective, but machine-virtualization technologies are now becoming the dominant way of running multiple operating systems on a single machine.

Windows 7 is a multiuser operating system, supporting simultaneous access through distributed services or through multiple instances of the GUI via the Windows terminal services. The server editions of Windows 7 support simultaneous terminal server sessions from Windows desktop systems. The desktop editions of terminal server multiplex the keyboard, mouse, and monitor between virtual terminal sessions for each logged-on user. This feature, called ***fast user switching***, allows users to preempt each other at the console of a PC without having to log off and log on.

We noted earlier that some GUI implementation moved into kernel mode in Windows NT 4.0. It started to move into user mode again with Windows Vista, which included the **desktop window manager** (**DWM**) as a user-mode process. DWM implements the desktop compositing of Windows, providing the Windows ***Aero*** interface look on top of the Windows DirectX graphic software. DirectX continues to run in the kernel, as does the code implementing Windows' previous windowing and graphics models (Win32k and GDI). Windows 7 made substantial changes to the DWM, significantly reducing its memory footprint and improving its performance.

**Windows XP** was the first version of Windows to ship a 64-bit version (for the IA64 in 2001 and the AMD64 in 2005). Internally, the native NT file system (NTFS) and many of the Win32 APIs have always used 64-bit integers where appropriate—so the major extension to 64-bit in Windows XP was support for large virtual addresses. However, 64-bit editions of Windows also support much larger physical memories. By the time Windows 7 shipped, the AMD64 ISA had become available on almost all CPUs from both Intel and AMD. In addition, by that time, physical memories on client systems frequently exceeded the 4-GB limit of the IA-32. As a result, the 64-bit version of Windows 7 is now commonly installed on larger client systems. Because the AMD64 architecture supports high-fidelity IA-32 compatibility at the level of individual processes, 32- and 64-bit applications can be freely mixed in a single system.

In the rest of our description of Windows 7, we will not distinguish between the client editions of Windows 7 and the corresponding server editions. They are based on the same core components and run the same binary files for the kernel and most drivers. Similarly, although Microsoft ships a variety of different editions of each release to address different market price points, few of the differences between editions are reflected in the core of the system. In this chapter, we focus primarily on the core components of Windows 7.

## B.2 Design Principles

Microsoft's design goals for Windows included security, reliability, Windows and POSIX application compatibility, high performance, extensibility, portability, and international support. Some additional goals, energy efficiency and dynamic device support, have recently been added to this list. Next, we discuss each of these goals and how it is achieved in Windows 7.

### B.2.1 Security

Windows 7 security goals required more than just adherence to the design standards that had enabled Windows NT 4.0 to receive a C2 security classification from the U.S. government. (A C2 classification signifies a moderate level of protection from defective software and malicious attacks. Classifications were defined by the Department of Defense Trusted Computer System Evaluation Criteria, also known as the **Orange Book**.) Extensive code review and testing were combined with sophisticated automatic analysis tools to identify and investigate potential defects that might represent security vulnerabilities.

Windows bases security on discretionary access controls. System objects, including files, registry settings, and kernel objects, are protected by **access-control lists** (**ACLs**) (see Section 13.4.2). ACLs are vulnerable to user and programmer errors, however, as well as to the most common attacks on consumer systems, in which the user is tricked into running code, often while browsing the web. Windows 7 includes a mechanism called **integrity levels** that acts as a rudimentary ***capability*** system for controlling access. Objects and processes are marked as having low, medium, or high integrity. Windows does not allow a process to modify an object with a higher integrity level, no matter what the setting of the ACL.

Other security measures include **address-space layout randomization** (**ASLR**), nonexecutable stacks and heaps, and encryption and **digital signature** facilities. ASLR thwarts many forms of attack by preventing small amounts of injected code from jumping easily to code that is already loaded in a process as part of normal operation. This safeguard makes it likely that a system under attack will fail or crash rather than let the attacking code take control.

Recent chips from both Intel and AMD are based on the AMD64 architecture, which allows memory pages to be marked so that they cannot contain executable instruction code. Windows tries to mark stacks and memory heaps so that they cannot be used to execute code, thus preventing attacks in which a program bug allows a buffer to overflow and then is tricked into executing the contents of the buffer. This technique cannot be applied to all programs, because some rely on modifying data and executing it. A column labeled “data execution prevention” in the Windows task manager shows which processes are marked to prevent these attacks.

Windows uses encryption as part of common protocols, such as those used to communicate securely with websites. Encryption is also used to protect user files stored on disk from prying eyes. Windows 7 allows users to easily encrypt virtually a whole disk, as well as removable storage devices such as USB flash drives, with a feature called BitLocker. If a computer with an encrypted disk is stolen, the thieves will need very sophisticated technology (such as an electron microscope) to gain access to any of the computer’s files. Windows uses digital signatures to ***sign*** operating system binaries so it can verify that the files were produced by Microsoft or another known company. In some editions of Windows, a **code integrity** module is activated at boot to ensure that all the loaded modules in the kernel have valid signatures, assuring that they have not been tampered with by an off-line attack.

### B.2.2 Reliability

Windows matured greatly as an operating system in its first ten years, leading to Windows 2000. At the same time, its reliability increased due to such factors as maturity in the source code, extensive stress testing of the system, improved CPU architectures, and automatic detection of many serious errors in drivers from both Microsoft and third parties. Windows has subsequently extended the tools for achieving reliability to include automatic analysis of source code for errors, tests that include providing invalid or unexpected input parameters (known as **fuzzing**) to detect validation failures, and an application version of the driver verifier that applies dynamic checking for an extensive set of common user-mode programming errors. Other improvements in reliability have resulted from moving more code out of the kernel and into user-mode services. Windows provides extensive support for writing drivers in user mode. System facilities that were once in the kernel and are now in user mode include the Desktop Window Manager and much of the software stack for audio.

One of the most significant improvements in the Windows experience came from adding memory diagnostics as an option at boot time. This addition is especially valuable because so few consumer PCs have error-correcting memory. When bad RAM starts to drop bits here and there, the result is frustratingly erratic behavior in the system. The availability of memory diagnostics has greatly reduced the stress levels of users with bad RAM.

Windows 7 introduced a fault-tolerant memory heap. The heap learns from application crashes and automatically inserts mitigations into future execution of an application that has crashed. This makes the application more reliable even if it contains common bugs such as using memory after freeing it or accessing past the end of the allocation.

Achieving high reliability in Windows is particularly challenging because almost one billion computers run Windows. Even reliability problems that affect only a small percentage of users still impact tremendous numbers of human beings. The complexity of the Windows ecosystem also adds to the challenges. Millions of instances of applications, drivers, and other software are being constantly downloaded and run on Windows systems. Of course, there is also a constant stream of malware attacks. As Windows itself has become harder to attack directly, exploits increasingly target popular applications.

To cope with these challenges, Microsoft is increasingly relying on communications from customer machines to collect large amounts of data from the ecosystem. Machines can be sampled to see how they are performing, what software they are running, and what problems they are encountering. Customers can send data to Microsoft when systems or software crashes or hangs. This constant stream of data from customer machines is collected very carefully, with the users' consent and without invading privacy. The result is that Microsoft is building an ever-improving picture of what is happening in the Windows ecosystem that allows continuous improvements through software updates, as well as providing data to guide future releases of Windows.

### B.2.3 Windows and POSIX Application Compatibility

As mentioned, Windows XP was both an update of Windows 2000 and a replacement for Windows 95/98. Windows 2000 focused primarily on compatibility for business applications. The requirements for Windows XP included a much greater compatibility with the consumer applications that ran on Windows 95/98. Application compatibility is difficult to achieve because many applications check for a particular version of Windows, may depend to some extent on the quirks of the implementation of APIs, may have latent application bugs that were masked in the previous system, and so forth. Applications may also have been compiled for a different instruction set. Windows 7 implements several strategies to run applications despite incompatibilities.

Like Windows XP, Windows 7 has a compatibility layer that sits between applications and the Win32 APIs. This layer makes Windows 7 look (almost) bug-for-bug compatible with previous versions of Windows. Windows 7, like earlier NT releases, maintains support for running many 16-bit applications using a ***thunking***, or conversion, layer that translates 16-bit API calls into equivalent 32-bit calls. Similarly, the 64-bit version of Windows 7 provides a thunking layer that translates 32-bit API calls into native 64-bit calls.

The Windows subsystem model allows multiple operating-system personalities to be supported. As noted earlier, although the API most commonly used with Windows is the Win32 API, some editions of Windows 7 support a POSIX subsystem. POSIX is a standard specification for UNIX that allows most available UNIX-compatible software to compile and run without modification.

As a final compatibility measure, several editions of Windows 7 provide a virtual machine that runs Windows XP inside Windows 7. This allows applications to get bug-for-bug compatibility with Windows XP.

### B.2.4 High Performance

Windows was designed to provide high performance on desktop systems (which are largely constrained by I/O performance), server systems (where the CPU is often the bottleneck), and large multithreaded and multiprocessor environments (where locking performance and cache-line management are keys to scalability). To satisfy performance requirements, NT used a variety of techniques, such as asynchronous I/O, optimized protocols for networks, kernel-based graphics rendering, and sophisticated caching of file-system data. The memory-management and synchronization algorithms were designed with an awareness of the performance considerations related to cache lines and multiprocessors.

Windows NT was designed for symmetrical multiprocessing (SMP); on a multiprocessor computer, several threads can run at the same time, even in the kernel. On each CPU, Windows NT uses priority-based preemptive scheduling of threads. Except while executing in the kernel dispatcher or at interrupt level, threads in any process running in Windows can be preempted by higher-priority threads. Thus, the system responds quickly (see Chapter 5).

The subsystems that constitute Windows NT communicate with one another efficiently through a **local procedure call** (**LPC**) facility that provides high-performance message passing. When a thread requests a synchronous service from another process through an LPC, the servicing thread is marked ***ready***, and its priority is temporarily boosted to avoid the scheduling delays that would occur if it had to wait for threads already in the queue.

Windows XP further improved performance by reducing the code-path length in critical functions, using better algorithms and per-processor data structures, using memory coloring for **non-uniform memory access** (**NUMA**) machines, and implementing more scalable locking protocols, such as queued spinlocks. The new locking protocols helped reduce system bus cycles and included lock-free lists and queues, atomic read–modify–write operations (like interlocked increment), and other advanced synchronization techniques.

By the time Windows 7 was developed, several major changes had come to computing. Client/server computing had increased in importance, so an advanced local procedure call (ALPC) facility was introduced to provide higher performance and more reliability than LPC. The number of CPUs and the amount of physical memory available in the largest multiprocessors had increased substantially, so quite a lot of effort was put into improving operating-system scalability.

The implementation of SMP in Windows NT used bitmasks to represent collections of processors and to identify, for example, which set of processors a particular thread could be scheduled on. These bitmasks were defined as fitting within a single word of memory, limiting the number of processors supported within a system to 64. Windows 7 added the concept of **processor groups** to represent arbitrary numbers of CPUs, thus accommodating more CPU cores. The number of CPU cores within single systems has continued to increase not only because of more cores but also because of cores that support more than one logical thread of execution at a time.

All these additional CPUs created a great deal of contention for the locks used for scheduling CPUs and memory. Windows 7 broke these locks apart. For example, before Windows 7, a single lock was used by the Windows scheduler to synchronize access to the queues containing threads waiting for events. In Windows 7, each object has its own lock, allowing the queues to be accessed concurrently. Also, many execution paths in the scheduler were rewritten to be lock-free. This change resulted in good scalability performance for Windows even on systems with 256 hardware threads.

Other changes are due to the increasing importance of support for parallel computing. For years, the computer industry has been dominated by Moore's Law, leading to higher densities of transistors that manifest themselves as faster clock rates for each CPU. Moore's Law continues to hold true, but limits have been reached that prevent CPU clock rates from increasing further. Instead, transistors are being used to build more and more CPUs into each chip. New programming models for achieving parallel execution, such as Microsoft's Concurrency RunTime (ConcRT) and Intel's Threading Building Blocks (TBB), are being used to express parallelism in C++ programs. Where Moore's Law has governed computing for forty years, it now seems that Amdahl's Law, which governs parallel computing, will rule the future.

To support task-based parallelism, Windows 7 provides a new form of **user-mode scheduling** (**UMS**). UMS allows programs to be decomposed into tasks, and the tasks are then scheduled on the available CPUs by a scheduler that operates in user mode rather than in the kernel.

The advent of multiple CPUs on the smallest computers is only part of the shift taking place to parallel computing. Graphics processing units (GPUs) accelerate the computational algorithms needed for graphics by using **SIMD** architectures to execute a single instruction for multiple data at the same time. This has given rise to the use of GPUs for general computing, not just graphics. Operating-system support for software like OpenCL and CUDA is allowing programs to take advantage of the GPUs. Windows supports use of GPUs through software in its DirectX graphics support. This software, called DirectCompute, allows programs to specify **computational kernels** using the same HLSL (high-level shader language) programming model used to program the SIMD hardware for **graphics shaders**. The computational kernels run very quickly on the GPU and return their results to the main computation running on the CPU.

### B.2.5 Extensibility

**Extensibility** refers to the capacity of an operating system to keep up with advances in computing technology. To facilitate change over time, the developers implemented Windows using a layered architecture. The Windows executive runs in kernel mode and provides the basic system services and abstractions that support shared use of the system. On top of the executive, several server subsystems operate in user mode. Among them are **environmental subsystems** that emulate different operating systems. Thus, programs written for the Win32 APIs and POSIX all run on Windows in the appropriate environment. Because of the modular structure, additional environmental subsystems can be added without affecting the executive. In addition, Windows uses loadable drivers in the I/O system, so new file systems, new kinds of I/O devices, and new kinds of networking can be added while the system is running. Windows uses a client–server model like the Mach operating system and supports distributed processing by **remote procedure calls** (**RPCs**) as defined by the Open Software Foundation.

### B.2.6 Portability

An operating system is **portable** if it can be moved from one CPU architecture to another with few changes. Windows was designed to be portable. Like the UNIX operating system, Windows is written primarily in C and C++. The architecture-specific source code is relatively small, and there is very little use of assembly code. Porting Windows to a new architecture mostly affects the Windows kernel, since the user-mode code in Windows is almost exclusively written to be architecture independent. To port Windows, the kernel's architecture-specific code must be ported, and sometimes conditional compilation is needed in other parts of the kernel because of changes in major data structures, such as the page-table format. The entire Windows system must then be recompiled for the new CPU instruction set.

Operating systems are sensitive not only to CPU architecture but also to CPU support chips and hardware boot programs. The CPU and support chips are collectively known as a **chipset**. These chipsets and the associated boot code determine how interrupts are delivered, describe the physical characteristics of each system, and provide interfaces to deeper aspects of the CPU architecture, such as error recovery and power management. It would be burdensome to have to port Windows to each type of support chip as well as to each CPU architecture. Instead, Windows isolates most of the chipset-dependent code in a dynamic link library (DLL), called the **hardware-abstraction layer** (**HAL**), that is loaded with the kernel. The Windows kernel depends on the HAL interfaces rather than on the underlying chipset details. This allows the single set of kernel and driver binaries for a particular CPU to be used with different chipsets simply by loading a different version of the HAL.

Over the years, Windows has been ported to a number of different CPU architectures: Intel IA-32-compatible 32-bit CPUs, AMD64-compatible and IA64 64-bit CPUs, the DEC Alpha, and the MIPS and PowerPC CPUs. Most of these CPU architectures failed in the market. When Windows 7 shipped, only the IA-

32 and AMD64 architectures were supported on client computers, along with AMD64 and IA64 on servers.

### B.2.7 International Support

Windows was designed for international and multinational use. It provides support for different locales via the **national-language-support** (**NLS**) API. The NLS API provides specialized routines to format dates, time, and money in accordance with national customs. String comparisons are specialized to account for varying character sets. UNICODE is Windows's native character code. Windows supports ANSI characters by converting them to UNICODE characters before manipulating them (8-bit to 16-bit conversion). System text strings are kept in resource files that can be replaced to localize the system for different languages. Multiple locales can be used concurrently, which is important to multilingual individuals and businesses.

### B.2.8 Energy Efficiency

Increasing energy efficiency for computers causes batteries to last longer for laptops and netbooks, saves significant operating costs for power and cooling of data centers, and contributes to green initiatives aimed at lowering energy consumption by businesses and consumers. For some time, Windows has implemented several strategies for decreasing energy use. The CPUs are moved to lower power states—for example, by lowering clock frequency—whenever possible. In addition, when a computer is not being actively used, Windows may put the entire computer into a low-power state (sleep) or may even save all of memory to disk and shut the computer off (hibernation). When the user returns, the computer powers up and continues from its previous state, so the user does not need to reboot and restart applications.

Windows 7 added some new strategies for saving energy. The longer a CPU can stay unused, the more energy can be saved. Because computers are so much faster than human beings, a lot of energy can be saved just while humans are thinking. The problem is that too many programs are constantly polling to see what is happening in the system. A swarm of software timers are firing, keeping the CPU from staying idle long enough to save much energy. Windows 7 extends CPU idle time by skipping clock ticks, coalescing software timers into smaller numbers of events, and "parking" entire CPUs when systems are not heavily loaded.

### B.2.9 Dynamic Device Support

Early in the history of the PC industry, computer configurations were fairly static. Occasionally, new devices might be plugged into the serial, printer, or game ports on the back of a computer, but that was it. The next steps toward dynamic configuration of PCs were laptop docks and PCMIA cards. A PC could suddenly be connected to or disconnected from a whole set of peripherals. In a contemporary PC, the situation has completely changed. PCs are designed to let users to plug and unplug a huge host of peripherals all the time; external disks, thumb drives, cameras, and the like are constantly coming and going.

Support for dynamic configuration of devices is continually evolving in Windows. The system can automatically recognize devices when they are

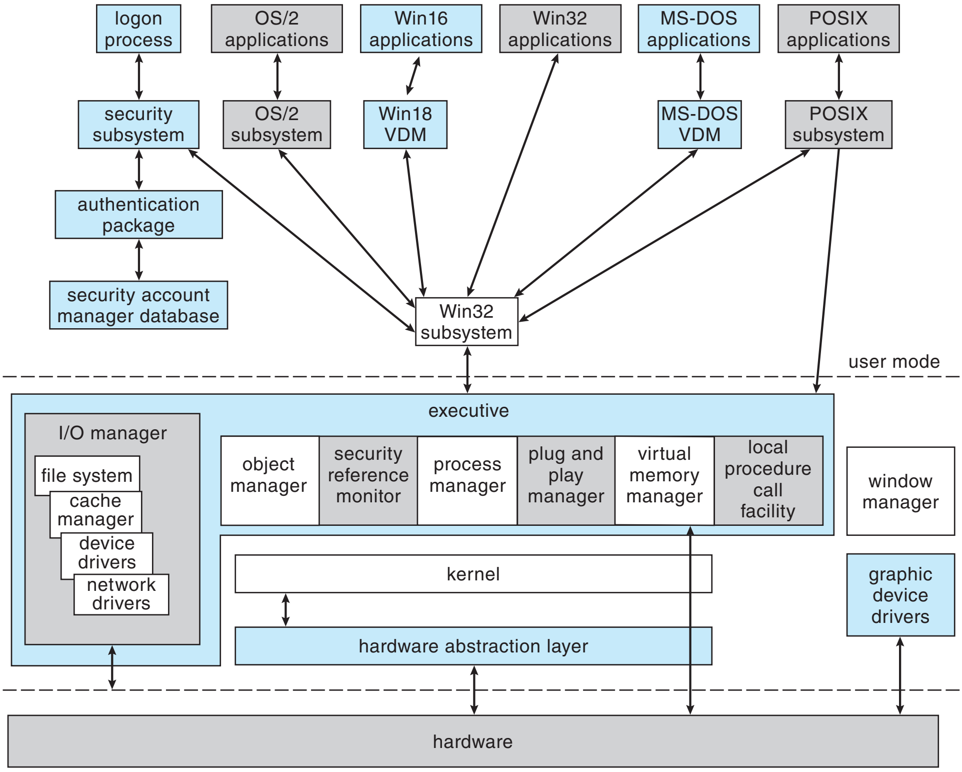

logon process
OS/2 applications
Win16 applications
Win32 applications
MS-DOS applications
POSIX applications
security subsystem
OS/2 subsystem
Win18 VDM
MS-DOS VDM
POSIX subsystem
authentication package
security account manager database
Win32 subsystem
user mode
executive
I/O manager
file system
cache manager
device drivers
network drivers
object manager
security reference monitor
process manager
plug and play manager
virtual memory manager
local procedure call facility
window manager
kernel
graphic device drivers
hardware abstraction layer
hardware

**Figure B.1** Windows block diagram.

plugged in and can find, install, and load the appropriate drivers—often without user intervention. When devices are unplugged, the drivers automatically unload, and system execution continues without disrupting other software.

## B.3 System Components

The architecture of Windows is a layered system of modules, as shown in Figure B.1. The main layers are the HAL, the kernel, and the executive, all of which run in kernel mode, and a collection of subsystems and services that run in user mode. The user-mode subsystems fall into two categories: the environmental subsystems, which emulate different operating systems, and the **protection subsystems**, which provide security functions. One of the chief advantages of this type of architecture is that interactions between modules are kept simple. The remainder of this section describes these layers and subsystems.

### B.3.1 Hardware-Abstraction Layer

The HAL is the layer of software that hides hardware chipset differences from upper levels of the operating system. The HAL exports a virtual hardware interface that is used by the kernel dispatcher, the executive, and the device drivers. Only a single version of each device driver is required for each CPU architecture, no matter what support chips might be present. Device drivers map devices and access them directly, but the chipset-specific details of mapping memory, configuring I/O buses, setting up DMA, and coping with motherboard-specific facilities are all provided by the HAL interfaces.

## B.3.2 Kernel

The kernel layer of Windows has four main responsibilities: thread scheduling, low-level processor synchronization, interrupt and exception handling, and switching between user mode and kernel mode. The kernel is implemented in the C language, using assembly language only where absolutely necessary to interface with the lowest level of the hardware architecture.

The kernel is organized according to object-oriented design principles. An **object type** in Windows is a system-defined data type that has a set of attributes (data values) and a set of methods (for example, functions or operations). An **object** is an instance of an object type. The kernel performs its job by using a set of kernel objects whose attributes store the kernel data and whose methods perform the kernel activities.

### B.3.2.1 Kernel Dispatcher

The kernel dispatcher provides the foundation for the executive and the subsystems. Most of the dispatcher is never paged out of memory, and its execution is never preempted. Its main responsibilities are thread scheduling and context switching, implementation of synchronization primitives, timer management, software interrupts (asynchronous and deferred procedure calls), and exception dispatching.

### B.3.2.2 Threads and Scheduling

Like many other modern operating systems, Windows uses processes and threads for executable code. Each process has one or more threads, and each thread has its own scheduling state, including actual priority, processor affinity, and CPU usage information.

There are six possible thread states: `ready`, `standby`, `running`, `waiting`, `transition`, and `terminated`. `Ready` indicates that the thread is waiting to run. The highest-priority ready thread is moved to the `standby` state, which means it is the next thread to run. In a multiprocessor system, each processor keeps one thread in a standby state. A thread is `running` when it is executing on a processor. It runs until it is preempted by a higher-priority thread, until it terminates, until its allotted execution time (quantum) ends, or until it waits on a dispatcher object, such as an event signaling I/O completion. A thread is in the `waiting` state when it is waiting for a dispatcher object to be signaled. A thread is in the `transition` state while it waits for resources necessary for execution; for example, it may be waiting for its kernel stack to be swapped in from disk. A thread enters the `terminated` state when it finishes execution.

The dispatcher uses a 32-level priority scheme to determine the order of thread execution. Priorities are divided into two classes: variable class and real-time class. The variable class contains threads having priorities from 1 to 15, and the real-time class contains threads with priorities ranging from 16 to 31.

The dispatcher uses a queue for each scheduling priority and traverses the set of queues from highest to lowest until it finds a thread that is ready to run. If a thread has a particular processor affinity but that processor is not available, the dispatcher skips past it and continues looking for a ready thread that is willing to run on the available processor. If no ready thread is found, the dispatcher executes a special thread called the ***idle thread***. Priority class 0 is reserved for the idle thread.

When a thread's time quantum runs out, the clock interrupt queues a quantum-end **deferred procedure call** (**DPC**) to the processor. Queuing the DPC results in a software interrupt when the processor returns to normal interrupt priority. The software interrupt causes the dispatcher to reschedule the processor to execute the next available thread at the preempted thread's priority level.

The priority of the preempted thread may be modified before it is placed back on the dispatcher queues. If the preempted thread is in the variable-priority class, its priority is lowered. The priority is never lowered below the base priority. Lowering the thread's priority tends to limit the CPU consumption of compute-bound threads versus I/O-bound threads. When a variable-priority thread is released from a wait operation, the dispatcher boosts the priority. The amount of the boost depends on the device for which the thread was waiting. For example, a thread waiting for keyboard I/O would get a large priority increase, whereas a thread waiting for a disk operation would get a moderate one. This strategy tends to give good response times to interactive threads using a mouse and windows. It also enables I/O-bound threads to keep the I/O devices busy while permitting compute-bound threads to use spare CPU cycles in the background. In addition, the thread associated with the user's active GUI window receives a priority boost to enhance its response time.

Scheduling occurs when a thread enters the ready or wait state, when a thread terminates, or when an application changes a thread's priority or processor affinity. If a higher-priority thread becomes ready while a lower-priority thread is running, the lower-priority thread is preempted. This preemption gives the higher-priority thread preferential access to the CPU. Windows is not a hard real-time operating system, however, because it does not guarantee that a real-time thread will start to execute within a particular time limit; threads are blocked indefinitely while DPCs and **interrupt service routines** (**ISRs**) are running (as discussed further below).

Traditionally, operating-system schedulers used sampling to measure CPU utilization by threads. The system timer would fire periodically, and the timer interrupt handler would take note of what thread was currently scheduled and whether it was executing in user or kernel mode when the interrupt occurred. This sampling technique was necessary because either the CPU did not have a high-resolution clock or the clock was too expensive or unreliable to access frequently. Although efficient, sampling was inaccurate and led to anomalies such as incorporating interrupt servicing time as thread time and dispatching threads that had run for only a fraction of the quantum. Starting with Windows Vista, CPU time in Windows has been tracked using the hardware **timestamp counter** (**TSC**) included in recent processors. Using the TSC results in more accurate accounting of CPU usage, and the scheduler will not preempt threads before they have run for a full quantum.

### B.3.2.3 Implementation of Synchronization Primitives

Key operating-system data structures are managed as objects using common facilities for allocation, reference counting, and security. **Dispatcher objects** control dispatching and synchronization in the system. Examples of these objects include the following:

- The **event object** is used to record an event occurrence and to synchronize this occurrence with some action. Notification events signal all waiting threads, and synchronization events signal a single waiting thread.
- The **mutant** provides kernel-mode or user-mode mutual exclusion associated with the notion of ownership.
- The **mutex**, available only in kernel mode, provides deadlock-free mutual exclusion.
- The **semaphore object** acts as a counter or gate to control the number of threads that access a resource.
- The **thread object** is the entity that is scheduled by the kernel dispatcher. It is associated with a **process object**, which encapsulates a virtual address space. The thread object is signaled when the thread exits, and the process object, when the process exits.
- The **timer object** is used to keep track of time and to signal timeouts when operations take too long and need to be interrupted or when a periodic activity needs to be scheduled.

Many of the dispatcher objects are accessed from user mode via an open operation that returns a handle. The user-mode code polls or waits on handles to synchronize with other threads as well as with the operating system (see Section B.7.1).

### B.3.2.4 Software Interrupts: Asynchronous and Deferred Procedure Calls

The dispatcher implements two types of software interrupts: **asynchronous procedure calls** (**APCs**) and deferred procedure calls (DPCs, mentioned earlier). An asynchronous procedure call breaks into an executing thread and calls a procedure. APCs are used to begin execution of new threads, suspend or resume existing threads, terminate threads or processes, deliver notification that an asynchronous I/O has completed, and extract the contents of the CPU registers from a running thread. APCs are queued to specific threads and allow the system to execute both system and user code within a process's context. User-mode execution of an APC cannot occur at arbitrary times, but only when the thread is waiting in the kernel and marked ***alertable***.

DPCsare used to postpone interrupt processing. After handling all urgent device-interrupt processing, the ISR schedules the remaining processing by queuing a DPC. The associated software interrupt will not occur until the CPU is next at a priority lower than the priority of all I/O device interrupts but higher than the priority at which threads run. Thus, DPCs do not block other device ISRs. In addition to deferring device-interrupt processing, the dispatcher uses

DPCs to process timer expirations and to preempt thread execution at the end of the scheduling quantum.

Execution of DPCs prevents threads from being scheduled on the current processor and also keeps APCs from signaling the completion of I/O. This is done so that completion of DPC routines does not take an extended amount of time. As an alternative, the dispatcher maintains a pool of worker threads. ISRs and DPCs may queue work items to the worker threads where they will be executed using normal thread scheduling. DPC routines are restricted so that they cannot take page faults (be paged out of memory), call system services, or take any other action that might result in an attempt to wait for a dispatcher object to be signaled. Unlike APCs, DPC routines make no assumptions about what process context the processor is executing.

### B.3.2.5 Exceptions and Interrupts

The kernel dispatcher also provides trap handling for exceptions and interrupts generated by hardware or software. Windows defines several architecture-independent exceptions, including:

- Memory-access violation
- Integer overflow
- Floating-point overflow or underflow
- Integer divide by zero
- Floating-point divide by zero
- Illegal instruction
- Data misalignment
- Privileged instruction
- Page-read error
- Access violation
- Paging file quota exceeded
- Debugger breakpoint
- Debugger single step

The trap handlers deal with simple exceptions. Elaborate exception handling is performed by the kernel's exception dispatcher. The **exception dispatcher** creates an exception record containing the reason for the exception and finds an exception handler to deal with it.

When an exception occurs in kernel mode, the exception dispatcher simply calls a routine to locate the exception handler. If no handler is found, a fatal system error occurs, and the user is left with the infamous "blue screen of death" that signifies system failure.

Exception handling is more complex for user-mode processes, because an environmental subsystem (such as the POSIX system) sets up a debugger port and an exception port for every process it creates. (For details on ports, see Section B.3.3.4.) If a debugger port is registered, the exception handler sends the exception to the port. If the debugger port is not found or does not handle that exception, the dispatcher attempts to find an appropriate exception handler. If no handler is found, the debugger is called again to catch the error for debugging. If no debugger is running, a message is sent to the process's exception port to give the environmental subsystem a chance to translate the exception. For example, the POSIX environment translates Windows exception messages into POSIX signals before sending them to the thread that caused the exception. Finally, if nothing else works, the kernel simply terminates the process containing the thread that caused the exception.

When Windows fails to handle an exception, it may construct a description of the error that occurred and request permission from the user to send the information back to Microsoft for further analysis. In some cases, Microsoft's automated analysis may be able to recognize the error immediately and suggest a fix or workaround.

The interrupt dispatcher in the kernel handles interrupts by calling either an interrupt service routine (ISR) supplied by a device driver or a kernel trap-handler routine. The interrupt is represented by an **interrupt object** that contains all the information needed to handle the interrupt. Using an interrupt object makes it easy to associate interrupt-service routines with an interrupt without having to access the interrupt hardware directly.

Different processor architectures have different types and numbers of interrupts. For portability, the interrupt dispatcher maps the hardware interrupts into a standard set. The interrupts are prioritized and are serviced in priority order. There are 32 interrupt request levels (IRQLs) in Windows. Eight are reserved for use by the kernel; the remaining 24 represent hardware interrupts via the HAL (although most IA-32 systems use only 16). The Windows interrupts are defined in Figure B.2.

The kernel uses an **interrupt-dispatch table** to bind each interrupt level to a service routine. In a multiprocessor computer, Windows keeps a separate interrupt-dispatch table (IDT) for each processor, and each processor's IRQL can be set independently to mask out interrupts. All interrupts that occur at a level equal to or less than the IRQL of a processor are blocked until the IRQL is lowered

| interrupt levels | types of interrupts |
|---|---|
| 31 | machine check or bus error |
| 30 | power fail |
| 29 | interprocessor notification (request another processor to act; e.g., dispatch a process or update the TLB) |
| 28 | clock (used to keep track of time) |
| 27 | profile |
| 3–26 | traditional PC IRQ hardware interrupts |
| 2 | dispatch and deferred procedure call (DPC) (kernel) |
| 1 | asynchronous procedure call (APC) |
| 0 | passive |

**Figure B.2** Windows interrupt-request levels.

by a kernel-level thread or by an ISR returning from interrupt processing. Windows takes advantage of this property and uses software interrupts to deliver APCs and DPCs, to perform system functions such as synchronizing threads with I/O completion, to start thread execution, and to handle timers.

#### B.3.2.6 Switching between User-Mode and Kernel-Mode Threads

What the programmer thinks of as a thread in traditional Windows is actually two threads: a **user-mode thread** (**UT**) and a **kernel-mode thread** (**KT**). Each has its own stack, register values, and execution context. A UT requests a system service by executing an instruction that causes a trap to kernel mode. The kernel layer runs a trap handler that switches between the UT and the corresponding KT. When a KT has completed its kernel execution and is ready to switch back to the corresponding UT, the kernel layer is called to make the switch to the UT, which continues its execution in user mode.

Windows 7 modifies the behavior of the kernel layer to support user-mode scheduling of the UTs. User-mode schedulers in Windows 7 support cooperative scheduling. A UT can explicitly yield to another UT by calling the user-mode scheduler; it is not necessary to enter the kernel. User-mode scheduling is explained in more detail in Section B.7.3.7.

### B.3.3 Executive

The Windows executive provides a set of services that all environmental subsystems use. The services are grouped as follows: object manager, virtual memory manager, process manager, advanced local procedure call facility, I/O manager, cache manager, security reference monitor, plug-and-play and power managers, registry, and booting.

#### B.3.3.1 Object Manager

For managing kernel-mode entities, Windows uses a generic set of interfaces that are manipulated by user-mode programs. Windows calls these entities ***objects***, and the executive component that manipulates them is the **object manager**. Examples of objects are semaphores, mutexes, events, processes, and threads; all these are ***dispatcher objects***. Threads can block in the kernel dispatcher waiting for any of these objects to be signaled. The process, thread, and virtual memory APIs use process and thread handles to identify the process or thread to be operated on. Other examples of objects include files, sections, ports, and various internal I/O objects. File objects are used to maintain the open state of files and devices. Sections are used to map files. Local-communication endpoints are implemented as port objects.

User-mode code accesses these objects using an opaque value called a **handle**, which is returned by many APIs. Each process has a **handle table** containing entries that track the objects used by the process. The **system process**, which contains the kernel, has its own handle table, which is protected from user code. The handle tables in Windows are represented by a tree structure, which can expand from holding 1,024 handles to holding over 16 million. Kernel-mode code can access an object by using either a handle or a **referenced pointer**.

A process gets a handle by creating an object, by opening an existing object, by receiving a duplicated handle from another process, or by inheriting a handle from the parent process. When a process exits, all its open handles are implicitly closed. Since the object manager is the only entity that generates object handles, it is the natural place to check security. The object manager checks whether a process has the right to access an object when the process tries to open the object. The object manager also enforces quotas, such as the maximum amount of memory a process may use, by charging a process for the memory occupied by all its referenced objects and refusing to allocate more memory when the accumulated charges exceed the process's quota.

The object manager keeps track of two counts for each object: the number of handles for the object and the number of referenced pointers. The handle count is the number of handles that refer to the object in the handle tables of all processes, including the system process that contains the kernel. The referenced pointer count is incremented whenever a new pointer is needed by the kernel and decremented when the kernel is done with the pointer. The purpose of these reference counts is to ensure that an object is not freed while it is still referenced by either a handle or an internal kernel pointer.

The object manager maintains the Windows internal name space. In contrast to UNIX, which roots the system name space in the file system, Windows uses an abstract name space and connects the file systems as devices. Whether a Windows object has a name is up to its creator. Processes and threads are created without names and referenced either by handle or through a separate numerical identifier. Synchronization events usually have names, so that they can be opened by unrelated processes. A name can be either permanent or temporary. A permanent name represents an entity, such as a disk drive, that remains even if no process is accessing it. A temporary name exists only while a process holds a handle to the object. The object manager supports directories and symbolic links in the name space. As an example, MS-DOS drive letters are implemented using symbolic links; `\Global??\C:` is a symbolic link to the device object `\Device\HarddiskVolume2`, representing a mounted file-system volume in the `\Device` directory.

Each object, as mentioned earlier, is an instance of an ***object type***. The object type specifies how instances are to be allocated, how the data fields are to be defined, and how the standard set of virtual functions used for all objects are to be implemented. The standard functions implement operations such as mapping names to objects, closing and deleting, and applying security checks. Functions that are specific to a particular type of object are implemented by system services designed to operate on that particular object type, not by the methods specified in the object type.

The `parse()` function is the most interesting of the standard object functions. It allows the implementation of an object. The file systems, the registry configuration store, and GUI objects are the most notable users of parse functions to extend the Windows name space.

Returning to our Windows naming example, device objects used to represent file-system volumes provide a parse function. This allows a name like `\Global??\C:\foo\bar.doc` to be interpreted as the file `\foo\bar.doc` on the volume represented by the device object `HarddiskVolume2`. We can illustrate how naming, parse functions, objects, and handles work together by looking at the steps to open the file in Windows:

1. An application requests that a file named `C:\foo\bar.doc` be opened.
2. The object manager finds the device object `HarddiskVolume2`, looks up the parse procedure `IopParseDevice` from the object's type, and invokes it with the file's name relative to the root of the file system.
3. `IopParseDevice()` allocates a file object and passes it to the file system, which fills in the details of how to access `C:\foo\bar.doc` on the volume.
4. When the file system returns, `IopParseDevice()` allocates an entry for the file object in the handle table for the current process and returns the handle to the application.

If the file cannot successfully be opened, `IopParseDevice()` deletes the file object it allocated and returns an error indication to the application.

#### B.3.3.2 Virtual Memory Manager

The executive component that manages the virtual address space, physical memory allocation, and paging is the **virtual memory (VM) manager**. The design of the VM manager assumes that the underlying hardware supports virtual-to-physical mapping, a paging mechanism, and transparent cache coherence on multiprocessor systems, as well as allowing multiple page-table entries to map to the same physical page frame. The VM manager in Windows uses a page-based management scheme with page sizes of 4 KB and 2 MB on AMD64 and IA-32-compatible processors and 8 KB on the IA64. Pages of data allocated to a process that are not in physical memory are either stored in the **paging file** on disk or mapped directly to a regular file on a local or remote file system. A page can also be marked zero-fill-on-demand, which initializes the page with zeros before it is allocated, thus erasing the previous contents.

On IA-32 processors, each process has a 4-GB virtual address space. The upper 2 GB are mostly identical for all processes and are used by Windows in kernel mode to access the operating-system code and data structures. For the AMD64 architecture, Windows provides a 8-TB virtual address space for user mode out of the 16 EB supported by existing hardware for each process.

Key areas of the kernel-mode region that are not identical for all processes are the self-map, hyperspace, and session space. The hardware references a process's page table using physical page-frame numbers, and the **page table self-map** makes the contents of the process's page table accessible using virtual addresses. **Hyperspace** maps the current process's working-set information into the kernel-mode address space. **Session space** is used to share an instance of the Win32 and other session-specific drivers among all the processes in the same terminal-server (TS) session. Different TS sessions share different instances of these drivers, yet they are mapped at the same virtual addresses. The lower, user-mode region of virtual address space is specific to each process and accessible by both user- and kernel-mode threads.

The Windows VM manager uses a two-step process to allocate virtual memory. The first step ***reserves*** one or more pages of virtual addresses in the process's virtual address space. The second step ***commits*** the allocation by assigning virtual memory space (physical memory or space in the paging files). Windows limits the amount of virtual memory space a process consumes by enforcing a quota on committed memory. A process decommits memory that it is no longer using to free up virtual memory space for use by other processes. The APIs used to reserve virtual addresses and commit virtual memory take a handle on a process object as a parameter. This allows one process to control the virtual memory of another. Environmental subsystems manage the memory of their client processes in this way.

Windows implements shared memory by defining a **section object**. After getting a handle to a section object, a process maps the memory of the section to a range of addresses, called a **view**. A process can establish a view of the entire section or only the portion it needs. Windows allows sections to be mapped not just into the current process but into any process for which the caller has a handle.

Sections can be used in many ways. A section can be backed by disk space either in the system-paging file or in a regular file (a **memory-mapped file**). A section can be ***based***, meaning that it appears at the same virtual address for all processes attempting to access it. Sections can also represent physical memory, allowing a 32-bit process to access more physical memory than can fit in its virtual address space. Finally, the memory protection of pages in the section can be set to read-only, read-write, read-write-execute, execute-only, no access, or copy-on-write.

Let's look more closely at the last two of these protection settings:

- A ***no-access page*** raises an exception if accessed. The exception can be used, for example, to check whether a faulty program iterates beyond the end of an array or simply to detect that the program attempted to access virtual addresses that are not committed to memory. User- and kernel-mode stacks use no-access pages as **guard pages** to detect stack overflows. Another use is to look for heap buffer overruns. Both the user-mode memory allocator and the special kernel allocator used by the device verifier can be configured to map each allocation onto the end of a page, followed by a no-access page to detect programming errors that access beyond the end of an allocation.
- The ***copy-on-write mechanism*** enables the VM manager to use physical memory more efficiently. When two processes want independent copies of data from the same section object, the VM manager places a single shared copy into virtual memory and activates the copy-on-write property for that region of memory. If one of the processes tries to modify data in a copy-on-write page, the VM manager makes a private copy of the page for the process.

The virtual address translation in Windows uses a multilevel page table. For IA-32 and AMD64 processors, each process has a **page directory** that contains 512 **page-directory entries** (**PDEs**) 8 bytes in size. Each PDE points to a **PTE table** that contains 512 **page-table entries** (**PTEs**) 8 bytes in size. Each PTE points to a 4-KB **page frame** in physical memory. For a variety of reasons, the hardware requires that the page directories or PTE tables at each level of a multilevel page table occupy a single page. Thus, the number of PDEs or PTEs that fit in a page determine how many virtual addresses are translated by that page. See Figure B.3 for a diagram of this structure.

The structure described so far can be used to represent only 1 GB of virtual address translation. For IA-32, a second page-directory level is needed, con-

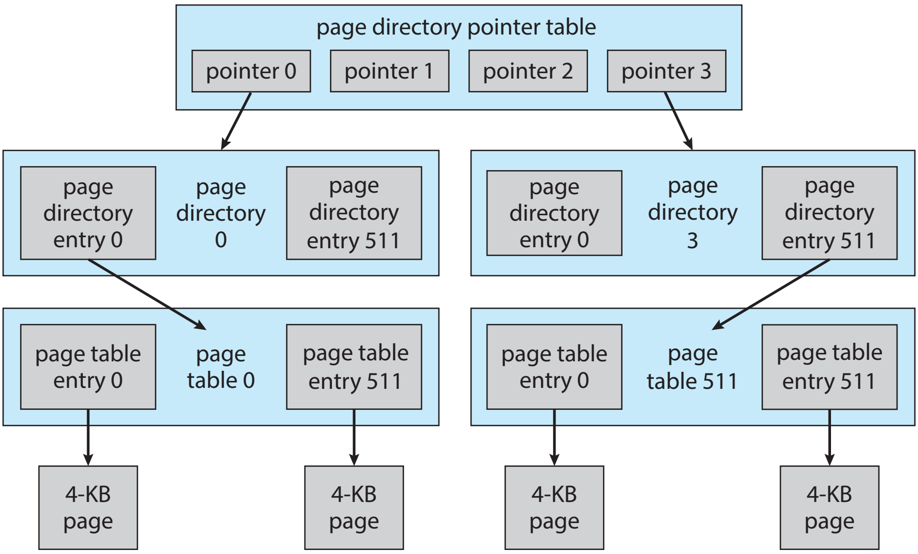

page directory pointer table
pointer 0
pointer 1
pointer 2
pointer 3
page directory entry 0
page directory 0
page directory entry 511
page directory entry 0
page directory 3
page directory entry 511
page table entry 0
page table 0
page table entry 511
page table entry 0
page table 511
page table entry 511
4-KB page
4-KB page
4-KB page
4-KB page

**Figure B.3** Page-table layout.

taining only four entries, as shown in the diagram. On 64-bit processors, more levels are needed. For AMD64, Windows uses a total of four full levels. The total size of all page-table pages needed to fully represent even a 32-bit virtual address space for a process is 8 MB. The VM manager allocates pages of PDEs and PTEs as needed and moves page-table pages to disk when not in use. The page-table pages are faulted back into memory when referenced.

We next consider how virtual addresses are translated into physical addresses on IA-32-compatible processors. A 2-bit value can represent the values 0, 1, 2, 3. A 9-bit value can represent values from 0 to 511; a 12-bit value, values from 0 to 4,095. Thus, a 12-bit value can select any byte within a 4-KB page of memory. A 9-bit value can represent any of the 512 PDEs or PTEs in a page directory or PTE-table page. As shown in Figure B.4, translating a virtual address pointer to a byte address in physical memory involves breaking the 32-bit pointer into four values, starting from the most significant bits:

- Two bits are used to index into the four PDEs at the top level of the page table. The selected PDE will contain the physical page number for each of the four page-directory pages that map 1 GB of the address space.

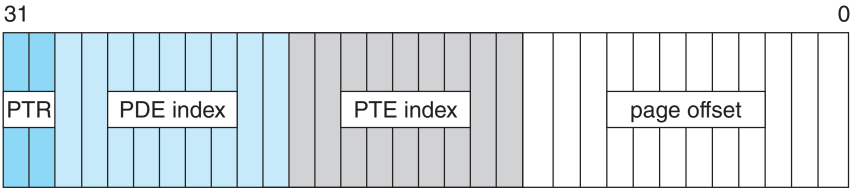

31
0
PTR
PDE index
PTE index
page offset

**Figure B.4** Virtual-to-physical address translation on IA-32.

- Nine bits are used to select another PDE, this time from a second-level page directory. This PDE will contain the physical page numbers of up to 512 PTE-table pages.
- Nine bits are used to select one of 512 PTEs from the selected PTE-table page. The selected PTE will contain the physical page number for the byte we are accessing.
- Twelve bits are used as the byte offset into the page. The physical address of the byte we are accessing is constructed by appending the lowest 12 bits of the virtual address to the end of the physical page number we found in the selected PTE.

The number of bits in a physical address may be different from the number of bits in a virtual address. In the original IA-32 architecture, the PTE and PDE were 32-bit structures that had room for only 20 bits of physical page number, so the physical address size and the virtual address size were the same. Such systems could address only 4 GB of physical memory. Later, the IA-32 was extended to the larger 64-bit PTE size used today, and the hardware supported 24-bit physical addresses. These systems could support 64 GB and were used on server systems. Today, all Windows servers are based on either the AMD64 or the IA64 and support very, very large physical addresses—more than we can possibly use. (Of course, once upon a time 4 GB seemed optimistically large for physical memory.)

To improve performance, the VM manager maps the page-directory and PTE-table pages into the same contiguous region of virtual addresses in every process. This self-map allows the VM manager to use the same pointer to access the current PDE or PTE corresponding to a particular virtual address no matter what process is running. The self-map for the IA-32 takes a contiguous 8-MB region of kernel virtual address space; the AMD64 self-map occupies 512 GB. Although the self-map occupies significant address space, it does not require any additional virtual memory pages. It also allows the page table's pages to be automatically paged in and out of physical memory.

In the creation of a self-map, one of the PDEs in the top-level page directory refers to the page-directory page itself, forming a "loop" in the page-table translations. The virtual pages are accessed if the loop is not taken, the PTE-table pages are accessed if the loop is taken once, the lowest-level page-directory pages are accessed if the loop is taken twice, and so forth.

The additional levels of page directories used for 64-bit virtual memory are translated in the same way except that the virtual address pointer is broken up into even more values. For the AMD64, Windows uses four full levels, each of which maps 512 pages, or 9+9+9+9+12 = 48 bits of virtual address.

To avoid the overhead of translating every virtual address by looking up the PDE and PTE, processors use **translation look-aside buffer** (**TLB**) hardware, which contains an associative memory cache for mapping virtual pages to PTEs. The TLB is part of the **memory-management unit** (**MMU**) within each processor. The MMU needs to "walk" (navigate the data structures of) the page table in memory only when a needed translation is missing from the TLB.

The PDEs and PTEs contain more than just physical page numbers. They also have bits reserved for operating-system use and bits that control how the hardware uses memory, such as whether hardware caching should be used for each page. In addition, the entries specify what kinds of access are allowed for both user and kernel modes.

A PDE can also be marked to say that it should function as a PTE rather than a PDE. On a IA-32, the first 11 bits of the virtual address pointer select a PDE in the first two levels of translation. If the selected PDE is marked to act as a PTE, then the remaining 21 bits of the pointer are used as the offset of the byte. This results in a 2-MB size for the page. Mixing and matching 4-KB and 2-MB page sizes within the page table is easy for the operating system and can significantly improve the performance of some programs by reducing how often the MMU needs to reload entries in the TLB, since one PDE mapping 2 MB replaces 512 PTEs each mapping 4 KB.

Managing physical memory so that 2-MB pages are available when needed is difficult, however, as they may continually be broken up into 4-KB pages, causing external fragmentation of memory. Also, the large pages can result in very significant internal fragmentation. Because of these problems, it is typically only Windows itself, along with large server applications, that use large pages to improve the performance of the TLB. They are better suited to do so because operating-system and server applications start running when the system boots, before memory has become fragmented.

Windows manages physical memory by associating each physical page with one of seven states: free, zeroed, modified, standby, bad, transition, or valid.

- A ***free*** page is a page that has no particular content.
- A ***zeroed*** page is a free page that has been zeroed out and is ready for immediate use to satisfy zero-on-demand faults.
- A ***modified*** page has been written by a process and must be sent to the disk before it is allocated for another process.
- A ***standby*** page is a copy of information already stored on disk. Standby pages may be pages that were not modified, modified pages that have already been written to the disk, or pages that were prefetched because they are expected to be used soon.
- A ***bad*** page is unusable because a hardware error has been detected.
- A ***transition*** page is on its way in from disk to a page frame allocated in physical memory.
- A ***valid*** page is part of the working set of one or more processes and is contained within these processes' page tables.

While valid pages are contained in processes' page tables, pages in other states are kept in separate lists according to state type. The lists are constructed by linking the corresponding entries in the **page frame number** (**PFN**) database, which includes an entry for each physical memory page. The PFN entries also include information such as reference counts, locks, and NUMA information. Note that the PFN database represents pages of physical memory, whereas the PTEs represent pages of virtual memory.

When the valid bit in a PTE is zero, hardware ignores all the other bits, and the VM manager can define them for its own use. Invalid pages can have a number of states represented by bits in the PTE. Page-file pages that have never

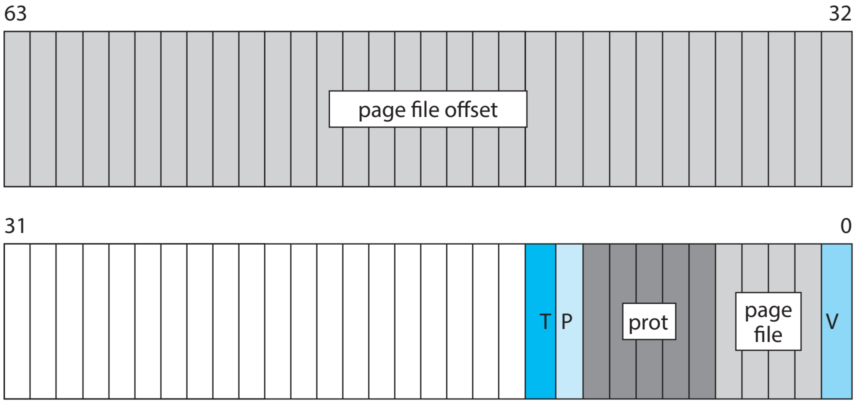


**Figure B.5** Page-file page-table entry. The valid bit is zero.

been faulted in are marked zero-on-demand. Pages mapped through section objects encode a pointer to the appropriate section object. PTEs for pages that have been written to the page file contain enough information to locate the page on disk, and so forth. The structure of the page-file PTE is shown in Figure B.5. The T, P, and V bits are all zero for this type of PTE. The PTE includes 5 bits for page protection, 32 bits for page-file offset, and 4 bits to select the paging file. There are also 20 bits reserved for additional bookkeeping.

Windows uses a per-working-set, least-recently-used (LRU) replacement policy to take pages from processes as appropriate. When a process is started, it is assigned a default minimum working-set size. The working set of each process is allowed to grow until the amount of remaining physical memory starts to run low, at which point the VM manager starts to track the age of the pages in each working set. Eventually, when the available memory runs critically low, the VM manager trims the working set to remove older pages.

How old a page is depends not on how long it has been in memory but on when it was last referenced. This is determined by periodically making a pass through the working set of each process and incrementing the age for pages that have not been marked in the PTE as referenced since the last pass. When it becomes necessary to trim the working sets, the VM manager uses heuristics to decide how much to trim from each process and then removes the oldest pages first.

A process can have its working set trimmed even when plenty of memory is available, if it was given a ***hard limit*** on how much physical memory it could use. In Windows 7, the VM manager will also trim processes that are growing rapidly, even if memory is plentiful. This policy change significantly improves the responsiveness of the system for other processes.

Windows tracks working sets not only for user-mode processes but also for the system process, which includes all the pageable data structures and code that run in kernel mode. Windows 7 created additional working sets for the system process and associated them with particular categories of kernel memory; the file cache, kernel heap, and kernel code now have their own working sets. The distinct working sets allow the VM manager to use different policies to trim the different categories of kernel memory.

The VM manager does not fault in only the page immediately needed. Research shows that the memory referencing of a thread tends to have a **locality** property. That is, when a page is used, it is likely that adjacent pages will be referenced in the near future. (Think of iterating over an array or fetching sequential instructions that form the executable code for a thread.) Because of locality, when the VM manager faults in a page, it also faults in a few adjacent pages. This prefetching tends to reduce the total number of page faults and allows reads to be clustered to improve I/O performance.

In addition to managing committed memory, the VM manager manages each process's reserved memory, or virtual address space. Each process has an associated tree that describes the ranges of virtual addresses in use and what the uses are. This allows the VM manager to fault in page-table pages as needed. If the PTE for a faulting address is uninitialized, the VM manager searches for the address in the process's tree of **virtual address descriptors** (**VADs**) and uses this information to fill in the PTE and retrieve the page. In some cases, a PTE-table page itself may not exist; such a page must be transparently allocated and initialized by the VM manager. In other cases, the page may be shared as part of a section object, and the VAD will contain a pointer to that section object. The section object contains information on how to find the shared virtual page so that the PTE can be initialized to point at it directly.

#### B.3.3.3 Process Manager

The Windows process manager provides services for creating, deleting, and using processes, threads, and jobs. It has no knowledge about parent–child relationships or process hierarchies; those refinements are left to the particular environmental subsystem that owns the process. The process manager is also not involved in the scheduling of processes, other than setting the priorities and affinities in processes and threads when they are created. Thread scheduling takes place in the kernel dispatcher.

Each process contains one or more threads. Processes themselves can be collected into larger units called **job objects**. The use of job objects allows limits to be placed on CPU usage, working-set size, and processor affinities that control multiple processes at once. Job objects are used to manage large data-center machines.

An example of process creation in the Win32 environment is as follows:

1. A Win32 application calls `CreateProcess()`.
2. A message is sent to the Win32 subsystem to notify it that the process is being created.
3. `CreateProcess()` in the original process then calls an API in the process manager of the NT executive to actually create the process.
4. The process manager calls the object manager to create a process object and returns the object handle to Win32.
5. Win32 calls the process manager again to create a thread for the process and returns handles to the new process and thread.

The Windows APIs for manipulating virtual memory and threads and for duplicating handles take a process handle, so subsystems can perform operations on behalf of a new process without having to execute directly in the new process's context. Once a new process is created, the initial thread is created, and an asynchronous procedure call is delivered to the thread to prompt the start of execution at the user-mode image loader. The loader is in `ntdll.dll`, which is a link library automatically mapped into every newly created process. Windows also supports a UNIX `fork()` style of process creation in order to support the POSIX environmental subsystem. Although the Win32 environment calls the process manager directly from the client process, POSIX uses the cross-process nature of the Windows APIs to create the new process from within the subsystem process.

The process manager relies on the asynchronous procedure calls (APCs) implemented by the kernel layer. APCs are used to initiate thread execution, suspend and resume threads, access thread registers, terminate threads and processes, and support debuggers.

The debugger support in the process manager includes the APIs to suspend and resume threads and to create threads that begin in suspended mode. There are also process-manager APIs that get and set a thread's register context and access another process's virtual memory. Threads can be created in the current process; they can also be injected into another process. The debugger makes use of thread injection to execute code within a process being debugged.

While running in the executive, a thread can temporarily attach to a different process. **Thread attach** is used by kernel worker threads that need to execute in the context of the process originating a work request. For example, the VM manager might use thread attach when it needs access to a process's working set or page tables, and the I/O manager might use it in updating the status variable in a process for asynchronous I/O operations.

The process manager also supports **impersonation**. Each thread has an associated **security token**. When the login process authenticates a user, the security token is attached to the user's process and inherited by its child processes. The token contains the **security identity** (**SID**) of the user, the SIDs of the groups the user belongs to, the privileges the user has, and the integrity level of the process. By default, all threads within a process share a common token, representing the user and the application that started the process. However, a thread running in a process with a security token belonging to one user can set a thread-specific token belonging to another user to impersonate that user.

The impersonation facility is fundamental to the client–server RPC model, where services must act on behalf of a variety of clients with different security IDs. The right to impersonate a user is most often delivered as part of an RPC connection from a client process to a server process. Impersonation allows the server to access system services as if it were the client in order to access or create objects and files on behalf of the client. The server process must be trustworthy and must be carefully written to be robust against attacks. Otherwise, one client could take over a server process and then impersonate any user who made a subsequent client request.

#### B.3.3.4 Facilities for Client–Server Computing

The implementation of Windows uses a client–server model throughout. The environmental subsystems are servers that implement particular operating-system personalities. Many other services, such as user authentication, network facilities, printer spooling, web services, network file systems, and plug-and-play, are also implemented using this model. To reduce the memory footprint, multiple services are often collected into a few processes running the `svchost.exe` program. Each service is loaded as a dynamic-link library (DLL), which implements the service by relying on the user-mode thread-pool facilities to share threads and wait for messages (see Section B.3.3.3).

The normal implementation paradigm for client–server computing is to use RPCs to communicate requests. The Win32 API supports a standard RPC protocol, as described in Section B.6.2.7. RPC uses multiple transports (for example, named pipes and TCP/IP) and can be used to implement RPCs between systems. When an RPC always occurs between a client and server on the local system, the advanced local procedure call facility (ALPC) can be used as the transport. At the lowest level of the system, in the implementation of the environmental systems, and for services that must be available in the early stages of booting, RPC is not available. Instead, native Windows services use ALPC directly.

ALPC is a message-passing mechanism. The server process publishes a globally visible connection-port object. When a client wants services from a subsystem or service, it opens a handle to the server's connection-port object and sends a connection request to the port. The server creates a channel and returns a handle to the client. The channel consists of a pair of private communication ports: one for client-to-server messages and the other for server-to-client messages. Communication channels support a callback mechanism, so the client and server can accept requests when they would normally be expecting a reply.

When an ALPC channel is created, one of three message-passing techniques is chosen.

1. The first technique is suitable for small to medium messages (up to 63 KB). In this case, the port's message queue is used as intermediate storage, and the messages are copied from one process to the other.
2. The second technique is for larger messages. In this case, a shared-memory section object is created for the channel. Messages sent through the port's message queue contain a pointer and size information referring to the section object. This avoids the need to copy large messages. The sender places data into the shared section, and the receiver views them directly.
3. The third technique uses APIs that read and write directly into a process's address space. ALPC provides functions and synchronization so that a server can access the data in a client. This technique is normally used by RPC to achieve higher performance for specific scenarios.

The Win32 window manager uses its own form of message passing, which is independent of the executive ALPC facilities. When a client asks for a connection that uses window-manager messaging, the server sets up three objects: (1) a dedicated server thread to handle requests, (2) a 64-KB shared section object, and (3) an event-pair object. An ***event-pair object*** is a synchronization object used by the Win32 subsystem to provide notification when the client thread has copied a message to the Win32 server, or vice versa. The section object is used to pass the messages, and the event-pair object provides synchronization.

Window-manager messaging has several advantages:

- The section object eliminates message copying, since it represents a region of shared memory.
- The event-pair object eliminates the overhead of using the port object to pass messages containing pointers and lengths.
- The dedicated server thread eliminates the overhead of determining which client thread is calling the server, since there is one server thread per client thread.
- The kernel gives scheduling preference to these dedicated server threads to improve performance.

#### B.3.3.5 I/O Manager

The **I/O manager** is responsible for managing file systems, device drivers, and network drivers. It keeps track of which device drivers, filter drivers, and file systems are loaded, and it also manages buffers for I/O requests. It works with the VM manager to provide memory-mapped file I/O and controls the Windows cache manager, which handles caching for the entire I/O system. The I/O manager is fundamentally asynchronous, providing synchronous I/O by explicitly waiting for an I/O operation to complete. The I/O manager provides several models of asynchronous I/O completion, including setting of events, updating of a status variable in the calling process, delivery of APCs to initiating threads, and use of I/O completion ports, which allow a single thread to process I/O completions from many other threads.

Device drivers are arranged in a list for each device (called a driver or I/O stack). A driver is represented in the system as a **driver object**. Because a single driver can operate on multiple devices, the drivers are represented in the I/O stack by a **device object**, which contains a link to the driver object. The I/O manager converts the requests it receives into a standard form called an **I/O request packet** (**IRP**). It then forwards the IRP to the first driver in the targeted I/O stack for processing. After a driver processes the IRP, it calls the I/O manager either to forward the IRP to the next driver in the stack or, if all processing is finished, to complete the operation represented by the IRP.

The I/O request may be completed in a context different from the one in which it was made. For example, if a driver is performing its part of an I/O operation and is forced to block for an extended time, it may queue the IRP to a worker thread to continue processing in the system context. In the original thread, the driver returns a status indicating that the I/O request is pending so that the thread can continue executing in parallel with the I/O operation. An IRP may also be processed in interrupt-service routines and completed in an arbitrary process context. Because some final processing may need to take place in the context that initiated the I/O, the I/O manager uses an APC to do final I/O-completion processing in the process context of the originating thread.

The I/O stack model is very flexible. As a driver stack is built, various drivers have the opportunity to insert themselves into the stack as **filte drivers**. Filter drivers can examine and potentially modify each I/O operation. Mount management, partition management, and disk striping and mirroring are all examples of functionality implemented using filter drivers that execute beneath the file system in the stack. File-system filter drivers execute above the file system and have been used to implement functionalities such as hierarchical storage management, single instancing of files for remote boot, and dynamic format conversion. Third parties also use file-system filter drivers to implement virus detection.

Device drivers for Windows are written to the Windows Driver Model (WDM) specification. This model lays out all the requirements for device drivers, including how to layer filter drivers, share common code for handling power and plug-and-play requests, build correct cancellation logic, and so forth.

Because of the richness of the WDM, writing a full WDM device driver for each new hardware device can involve a great deal of work. Fortunately, the port/miniport model makes it unnecessary to do this. Within a class of similar devices, such as audio drivers, SATA devices, or Ethernet controllers, each instance of a device shares a common driver for that class, called a **port driver**. The port driver implements the standard operations for the class and then calls device-specific routines in the device's **miniport driver** to implement device-specific functionality. The TCP/IP network stack is implemented in this way, with the `ndis.sys` class driver implementing much of the network driver functionality and calling out to the network miniport drivers for specific hardware.

Recent versions of Windows, including Windows 7, provide additional simplifications for writing device drivers for hardware devices. Kernel-mode drivers can now be written using the **Kernel-Mode Driver Framework** (**KMDF**), which provides a simplified programming model for drivers on top of WDM. Another option is the **User-Mode Driver Framework** (**UMDF**). Many drivers do not need to operate in kernel mode, and it is easier to develop and deploy drivers in user mode. It also makes the system more reliable, because a failure in a user-mode driver does not cause a kernel-mode crash.

#### B.3.3.6 Cache Manager

In many operating systems, caching is done by the file system. Instead, Windows provides a centralized caching facility. The **cache manager** works closely with the VM manager to provide cache services for all components under the control of the I/O manager. Caching in Windows is based on files rather than raw blocks. The size of the cache changes dynamically according to how much free memory is available in the system. The cache manager maintains a private working set rather than sharing the system process's working set. The cache manager memory-maps files into kernel memory and then uses special interfaces to the VM manager to fault pages into or trim them from this private working set.

The cache is divided into blocks of 256 KB. Each cache block can hold a view (that is, a memory-mapped region) of a file. Each cache block is described by a **virtual address control block** (**VACB**) that stores the virtual address and file offset for the view, as well as the number of processes using the view. The VACBs reside in a single array maintained by the cache manager.

When the I/O manager receives a file's user-level read request, the I/O manager sends an IRP to the I/O stack for the volume on which the file resides. For files that are marked as cacheable, the file system calls the cache manager

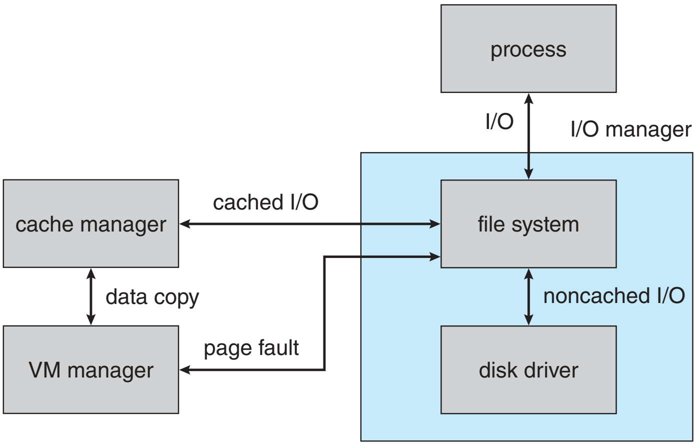


**Figure B.6** File I/O.

to look up the requested data in its cached file views. The cache manager calculates which entry of that file's VACB index array corresponds to the byte offset of the request. The entry either points to the view in the cache or is invalid. If it is invalid, the cache manager allocates a cache block (and the corresponding entry in the VACB array) and maps the view into the cache block. The cache manager then attempts to copy data from the mapped file to the caller's buffer. If the copy succeeds, the operation is completed.

If the copy fails, it does so because of a page fault, which causes the VM manager to send a noncached read request to the I/O manager. The I/O manager sends another request down the driver stack, this time requesting a paging operation, which bypasses the cache manager and reads the data from the file directly into the page allocated for the cache manager. Upon completion, the VACB is set to point at the page. The data, now in the cache, are copied to the caller's buffer, and the original I/O request is completed. Figure B.6 shows an overview of these operations.

A kernel-level read operation is similar, except that the data can be accessed directly from the cache rather than being copied to a buffer in user space. To use file-system metadata (data structures that describe the file system), the kernel uses the cache manager's mapping interface to read the metadata. To modify the metadata, the file system uses the cache manager's pinning interface. **Pinning** a page locks the page into a physical-memory page frame so that the VM manager cannot move the page or page it out. After updating the metadata, the file system asks the cache manager to unpin the page. A modified page is marked dirty, and so the VM manager flushes the page to disk.

To improve performance, the cache manager keeps a small history of read requests and from this history attempts to predict future requests. If the cache manager finds a pattern in the previous three requests, such as sequential access forward or backward, it prefetches data into the cache before the next request is submitted by the application. In this way, the application may find its data already cached and not need to wait for disk I/O.

The cache manager is also responsible for telling the VM manager to flush the contents of the cache. The cache manager's default behavior is write-back caching: it accumulates writes for 4 to 5 seconds and then wakes up the cache-writer thread. When write-through caching is needed, a process can set a flag when opening the file, or the process can call an explicit cache-flush function.

A fast-writing process could potentially fill all the free cache pages before the cache-writer thread had a chance to wake up and flush the pages to disk. The cache writer prevents a process from flooding the system in the following way. When the amount of free cache memory becomes low, the cache manager temporarily blocks processes attempting to write data and wakes the cache-writer thread to flush pages to disk. If the fast-writing process is actually a network redirector for a network file system, blocking it for too long could cause network transfers to time out and be retransmitted. This retransmission would waste network bandwidth. To prevent such waste, network redirectors can instruct the cache manager to limit the backlog of writes in the cache.

Because a network file system needs to move data between a disk and the network interface, the cache manager also provides a DMA interface to move the data directly. Moving data directly avoids the need to copy data through an intermediate buffer.

#### B.3.3.7 Security Reference Monitor

Centralizing management of system entities in the object manager enables Windows to use a uniform mechanism to perform run-time access validation and audit checks for every user-accessible entity in the system. Whenever a process opens a handle to an object, the **security reference monitor** (**SRM**) checks the process's security token and the object's access-control list to see whether the process has the necessary access rights.

The SRM is also responsible for manipulating the privileges in security tokens. Special privileges are required for users to perform backup or restore operations on file systems, debug processes, and so forth. Tokens can also be marked as being restricted in their privileges so that they cannot access objects that are available to most users. Restricted tokens are used primarily to limit the damage that can be done by execution of untrusted code.

The integrity level of the code executing in a process is also represented by a token. Integrity levels are a type of capability mechanism, as mentioned earlier. A process cannot modify an object with an integrity level higher than that of the code executing in the process, whatever other permissions have been granted. Integrity levels were introduced to make it harder for code that successfully attacks outward-facing software, like Internet Explorer, to take over a system.

Another responsibility of the SRM is logging security audit events. The Department of Defense's **Common Criteria** (the 2005 successor to the Orange Book) requires that a secure system have the ability to detect and log all attempts to access system resources so that it can more easily trace attempts at unauthorized access. Because the SRM is responsible for making access checks, it generates most of the audit records in the security-event log.

#### B.3.3.8 Plug-and-Play Manager

The operating system uses the **plug-and-play** (**PnP**) manager to recognize and adapt to changes in the hardware configuration. PnP devices use standard protocols to identify themselves to the system. The PnP manager automatically recognizes installed devices and detects changes in devices as the system operates. The manager also keeps track of hardware resources used by a device, as well as potential resources that could be used, and takes care of loading the appropriate drivers. This management of hardware resources—primarily interrupts and I/O memory ranges—has the goal of determining a hardware configuration in which all devices are able to operate successfully.

The PnP manager handles dynamic reconfiguration as follows. First, it gets a list of devices from each bus driver (for example, PCI or USB). It loads the installed driver (after finding one, if necessary) and sends an `add-device` request to the appropriate driver for each device. The PnP manager then figures out the optimal resource assignments and sends a `start-device` request to each driver specifying the resource assignments for the device. If a device needs to be reconfigured, the PnP manager sends a `query-stop` request, which asks the driver whether the device can be temporarily disabled. If the driver can disable the device, then all pending operations are completed, and new operations are prevented from starting. Finally, the PnP manager sends a `stop` request and can then reconfigure the device with a new `start-device` request.

The PnP manager also supports other requests. For example, `query-remove`, which operates similarly to `query-stop`, is employed when a user is getting ready to eject a removable device, such as a USB storage device. The `surprise-remove` request is used when a device fails or, more likely, when a user removes a device without telling the system to stop it first. Finally, the `remove` request tells the driver to stop using a device permanently.

Many programs in the system are interested in the addition or removal of devices, so the PnP manager supports notifications. Such a notification, for example, gives GUI file menus the information they need to update their list of disk volumes when a new storage device is attached or removed. Installing devices often results in adding new services to the `svchost.exe` processes in the system. These services frequently set themselves up to run whenever the system boots and continue to run even if the original device is never plugged into the system. Windows 7 introduced a **service-trigger** mechanism in the **service control manager** (**SCM**), which manages the system services. With this mechanism, services can register themselves to start only when SCM receives a notification from the PnP manager that the device of interest has been added to the system.

#### B.3.3.9 Power Manager

Windows works with the hardware to implement sophisticated strategies for energy efficiency, as described in Section B.2.8. The policies that drive these strategies are implemented by the **power manager**. The power manager detects current system conditions, such as the load on CPUs or I/O devices, and improves energy efficiency by reducing the performance and responsiveness of the system when need is low. The power manager can also put the entire system into a very efficient ***sleep*** mode and can even write all the contents of memory to disk and turn off the power to allow the system to go into ***hibernation***.

The primary advantage of sleep is that the system can enter fairly quickly, perhaps just a few seconds after the lid closes on a laptop. The return from sleep is also fairly quick. The power is turned down low on the CPUs and I/O devices, but the memory continues to be powered enough that its contents are not lost.

Hibernation takes considerably longer because the entire contents of memory must be transferred to disk before the system is turned off. However, the fact that the system is, in fact, turned off is a significant advantage. If there is a loss of power to the system, as when the battery is swapped on a laptop or a desktop system is unplugged, the saved system data will not be lost. Unlike shutdown, hibernation saves the currently running system so a user can resume where she left off, and because hibernation does not require power, a system can remain in hibernation indefinitely.

Like the PnP manager, the power manager provides notifications to the rest of the system about changes in the power state. Some applications want to know when the system is about to be shut down so they can start saving their states to disk.

### B.3.3.10 Registry

Windows keeps much of its configuration information in internal databases, called **hives**, that are managed by the Windows configuration manager, which is commonly known as the **registry**. There are separate hives for system information, default user preferences, software installation, security, and boot options. Because the information in the **system hive** is required to boot the system, the registry manager is implemented as a component of the executive.

The registry represents the configuration state in each hive as a hierarchical namespace of keys (directories), each of which can contain a set of typed values, such as UNICODE string, ANSI string, integer, or untyped binary data. In theory, new keys and values are created and initialized as new software is installed; then they are modified to reflect changes in the configuration of that software. In practice, the registry is often used as a general-purpose database, as an interprocess-communication mechanism, and for many other such inventive purposes.

Restarting applications, or even the system, every time a configuration change was made would be a nuisance. Instead, programs rely on various kinds of notifications, such as those provided by the PnP and power managers, to learn about changes in the system configuration. The registry also supplies notifications; it allows threads to register to be notified when changes are made to some part of the registry. The threads can thus detect and adapt to configuration changes recorded in the registry itself.

Whenever significant changes are made to the system, such as when updates to the operating system or drivers are installed, there is a danger that the configuration data may be corrupted (for example, if a working driver is replaced by a nonworking driver or an application fails to install correctly and leaves partial information in the registry). Windows creates a **system restore point** before making such changes. The restore point contains a copy of the hives before the change and can be used to return to this version of the hives and thereby get a corrupted system working again.

To improve the stability of the registry configuration, Windows added a transaction mechanism beginning with Windows Vista that can be used to prevent the registry from being partially updated with a collection of related configuration changes. Registry transactions can be part of more general transactions administered by the **kernel transaction manager** (**KTM**), which can also include file-system transactions. KTM transactions do not have the full semantics found in normal database transactions, and they have not supplanted the system restore facility for recovering from damage to the registry configuration caused by software installation.

#### B.3.3.11 Booting

The booting of a Windows PC begins when the hardware powers on and firmware begins executing from ROM. In older machines, this firmware was known as the BIOS, but more modern systems use UEFI (the Unified Extensible Firmware Interface), which is faster and more general and makes better use of the facilities in contemporary processors. The firmware runs **power-on self-test** (**POST**) diagnostics; identifies many of the devices attached to the system and initializes them to a clean, power-up state; and then builds the description used by the **advanced configuratio and power interface** (**ACPI**). Next, the firmware finds the system disk, loads the Windows `bootmgr` program, and begins executing it.

In a machine that has been hibernating, the `winresume` program is loaded next. It restores the running system from disk, and the system continues execution at the point it had reached right before hibernating. In a machine that has been shut down, the `bootmgr` performs further initialization of the system and then loads `winload`. This program loads `hal.dll`, the kernel (`ntoskrnl.exe`), any drivers needed in booting, and the system hive. `winload` then transfers execution to the kernel.

The kernel initializes itself and creates two processes. The **system process** contains all the internal kernel worker threads and never executes in user mode. The first user-mode process created is SMSS, for ***session manager subsystem***, which is similar to the INIT (initialization) process in UNIX. SMSS performs further initialization of the system, including establishing the paging files, loading more device drivers, and managing the Windows sessions. Each session is used to represent a logged-on user, except for ***session 0***, which is used to run system-wide background services, such as LSASS and SERVICES. A session is anchored by an instance of the CSRSS process. Each session other than 0 initially runs the WINLOGON process. This process logs on a user and then launches the EXPLORER process, which implements the Windows GUI experience. The following list itemizes some of these aspects of booting:

- SMSS completes system initialization and then starts up session 0 and the first login session.
- WININIT runs in session 0 to initialize user mode and start LSASS, SERVICES, and the local session manager, LSM.
- LSASS, the security subsystem, implements facilities such as authentication of users.
- SERVICES contains the service control manager, or SCM, which supervises all background activities in the system, including user-mode services. A number of services will have registered to start when the system boots. Others will be started only on demand or when triggered by an event such as the arrival of a device.

- CSRSS is the Win32 environmental subsystem process. It is started in every session—unlike the POSIX subsystem, which is started only on demand when a POSIX process is created.
- WINLOGON is run in each Windows session other than session 0 to log on a user.

The system optimizes the boot process by prepaging from files on disk based on previous boots of the system. Disk access patterns at boot are also used to lay out system files on disk to reduce the number of I/O operations required. The processes necessary to start the system are reduced by grouping services into fewer processes. All of these approaches contribute to a dramatic reduction in system boot time. Of course, system boot time is less important than it once was because of the sleep and hibernation capabilities of Windows.

## B.4 Terminal Services and Fast User Switching

Windows supports a GUI-based console that interfaces with the user via keyboard, mouse, and display. Most systems also support audio and video. Audio input is used by Windows voice-recognition software; voice recognition makes the system more convenient and increases its accessibility for users with disabilities. Windows 7 added support for **multi-touch hardware**, allowing users to input data by touching the screen and making gestures with one or more fingers. Eventually, the video-input capability, which is currently used for communication applications, is likely to be used for visually interpreting gestures, as Microsoft has demonstrated for its Xbox 360 Kinect product. Other future input experiences may evolve from Microsoft's **surface computer**. Most often installed at public venues, such as hotels and conference centers, the surface computer is a table surface with special cameras underneath. It can track the actions of multiple users at once and recognize objects that are placed on top.

The PC was, of course, envisioned as a ***personal computer***—an inherently single-user machine. Modern Windows, however, supports the sharing of a PC among multiple users. Each user that is logged on using the GUI has a **session** created to represent the GUI environment he will be using and to contain all the processes created to run his applications. Windows allows multiple sessions to exist at the same time on a single machine. However, Windows only supports a single console, consisting of all the monitors, keyboards, and mice connected to the PC. Only one session can be connected to the console at a time. From the logon screen displayed on the console, users can create new sessions or attach to an existing session that was previously created. This allows multiple users to share a single PC without having to log off and on between users. Microsoft calls this use of sessions ***fast user switching***.

Users can also create new sessions, or connect to existing sessions, on one PC from a session running on another Windows PC. The terminal server (TS) connects one of the GUI windows in a user's local session to the new or existing session, called a **remote desktop**, on the remote computer. The most common use of remote desktops is for users to connect to a session on their work PC from their home PC.

Many corporations use corporate terminal-server systems maintained in data centers to run all user sessions that access corporate resources, rather than allowing users to access those resources from the PCs in each user's office. Each server computer may handle many dozens of remote-desktop sessions. This is a form of **thin-client** computing, in which individual computers rely on a server for many functions. Relying on data-center terminal servers improves reliability, manageability, and security of the corporate computing resources.

The TS is also used by Windows to implement **remote assistance**. A remote user can be invited to share a session with the user logged on to the session on the console. The remote user can watch the user's actions and even be given control of the desktop to help resolve computing problems.

## B.5 File System

The native file system in Windows is NTFS. It is used for all local volumes. However, associated USB thumb drives, flash memory on cameras, and external disks may be formatted with the 32-bit FAT file system for portability. FAT is a much older file-system format that is understood by many systems besides Windows, such as the software running on cameras. A disadvantage is that the FAT file system does not restrict file access to authorized users. The only solution for securing data with FAT is to run an application to encrypt the data before storing it on the file system.

In contrast, NTFS uses ACLs to control access to individual files and supports implicit encryption of individual files or entire volumes (using Windows BitLocker feature). NTFS implements many other features as well, including data recovery, fault tolerance, very large files and file systems, multiple data streams, UNICODE names, sparse files, journaling, volume shadow copies, and file compression.

### B.5.1 NTFS Internal Layout

The fundamental entity in NTFS is a volume. A volume is created by the Windows logical disk management utility and is based on a logical disk partition. A volume may occupy a portion of a disk or an entire disk, or may span several disks.

NTFS does not deal with individual sectors of a disk but instead uses clusters as the units of disk allocation. A **cluster** is a number of disk sectors that is a power of 2. The cluster size is configured when an NTFS file system is formatted. The default cluster size is based on the volume size—4 KB for volumes larger than 2 GB. Given the size of today's disks, it may make sense to use cluster sizes larger than the Windows defaults to achieve better performance, although these performance gains will come at the expense of more internal fragmentation.

NTFS uses **logical cluster numbers** (**LCNs**) as disk addresses. It assigns them by numbering clusters from the beginning of the disk to the end. Using this scheme, the system can calculate a physical disk offset (in bytes) by multiplying the LCN by the cluster size.

A file in NTFS is not a simple byte stream as it is in UNIX; rather, it is a structured object consisting of typed **attributes**. Each attribute of a file is an independent byte stream that can be created, deleted, read, and written. Some attribute types are standard for all files, including the file name (or names, if the file has aliases, such as an MS-DOS short name), the creation time, and the security descriptor that specifies the access control list. User data are stored in ***data attributes***.

Most traditional data files have an ***unnamed*** data attribute that contains all the file's data. However, additional data streams can be created with explicit names. For instance, in Macintosh files stored on a Windows server, the resource fork is a named data stream. The IProp interfaces of the Component Object Model (COM) use a named data stream to store properties on ordinary files, including thumbnails of images. In general, attributes may be added as necessary and are accessed using a ***file-name:attribute*** syntax. NTFS returns only the size of the unnamed attribute in response to file-query operations, such as when running the `dir` command.

Every file in NTFS is described by one or more records in an array stored in a special file called the master file table (MFT). The size of a record is determined when the file system is created; it ranges from 1 to 4 KB. Small attributes are stored in the MFT record itself and are called ***resident attributes***. Large attributes, such as the unnamed bulk data, are called ***nonresident attributes*** and are stored in one or more contiguous **extents** on the disk. A pointer to each extent is stored in the MFT record. For a small file, even the data attribute may fit inside the MFT record. If a file has many attributes—or if it is highly fragmented, so that many pointers are needed to point to all the fragments—one record in the MFT might not be large enough. In this case, the file is described by a record called the **base file record**, which contains pointers to overflow records that hold the additional pointers and attributes.

Each file in an NTFS volume has a unique ID called a **file reference**. The file reference is a 64-bit quantity that consists of a 48-bit file number and a 16-bit sequence number. The file number is the record number (that is, the array slot) in the MFT that describes the file. The sequence number is incremented every time an MFT entry is reused. The sequence number enables NTFS to perform internal consistency checks, such as catching a stale reference to a deleted file after the MFT entry has been reused for a new file.

#### B.5.1.1 NTFS B+ Tree

As in UNIX, the NTFS namespace is organized as a hierarchy of directories. Each directory uses a data structure called a **B+ tree** to store an index of the file names in that directory. In a B+ tree, the length of every path from the root of the tree to a leaf is the same, and the cost of reorganizing the tree is eliminated. The **index root** of a directory contains the top level of the B+ tree. For a large directory, this top level contains pointers to disk extents that hold the remainder of the tree. Each entry in the directory contains the name and file reference of the file, as well as a copy of the update timestamp and file size taken from the file's resident attributes in the MFT. Copies of this information are stored in the directory so that a directory listing can be efficiently generated. Because all the file names, sizes, and update times are available from the directory itself, there is no need to gather these attributes from the MFT entries for each of the files.

#### B.5.1.2 NTFS Metadata

The NTFS volume's metadata are all stored in files. The first file is the MFT. The second file, which is used during recovery if the MFT is damaged, contains a copy of the first 16 entries of the MFT. The next few files are also special in purpose. They include the files described below.

- The log file records all metadata updates to the file system.
- The **volume file** contains the name of the volume, the version of NTFS that formatted the volume, and a bit that tells whether the volume may have been corrupted and needs to be checked for consistency using the `chkdsk` program.
- The **attribute-definition table** indicates which attribute types are used in the volume and what operations can be performed on each of them.
- The **root directory** is the top-level directory in the file-system hierarchy.
- The **bitmap file** indicates which clusters on a volume are allocated to files and which are free.
- The **boot file** contains the startup code for Windows and must be located at a particular disk address so that it can be found easily by a simple ROM bootstrap loader. The boot file also contains the physical address of the MFT.
- The **bad-cluster file** keeps track of any bad areas on the volume; NTFS uses this record for error recovery.

Keeping all the NTFS metadata in actual files has a useful property. As discussed in Section B.3.3.6, the cache manager caches file data. Since all the NTFS metadata reside in files, these data can be cached using the same mechanisms used for ordinary data.

### B.5.2 Recovery

In many simple file systems, a power failure at the wrong time can damage the file-system data structures so severely that the entire volume is scrambled. Many UNIX file systems, including UFS but not ZFS, store redundant metadata on the disk, and they recover from crashes by using the `fsck` program to check all the file-system data structures and restore them forcibly to a consistent state. Restoring them often involves deleting damaged files and freeing data clusters that had been written with user data but not properly recorded in the file system's metadata structures. This checking can be a slow process and can cause the loss of significant amounts of data.

NTFS takes a different approach to file-system robustness. In NTFS, all file-system data-structure updates are performed inside transactions. Before a data structure is altered, the transaction writes a log record that contains redo and undo information. After the data structure has been changed, the transaction writes a commit record to the log to signify that the transaction succeeded.

After a crash, the system can restore the file-system data structures to a consistent state by processing the log records, first redoing the operations for committed transactions and then undoing the operations for transactions that did not commit successfully before the crash. Periodically (usually every 5 seconds), a checkpoint record is written to the log. The system does not need log records prior to the checkpoint to recover from a crash. They can be discarded, so the log file does not grow without bounds. The first time after system startup that an NTFS volume is accessed, NTFS automatically performs file-system recovery.

This scheme does not guarantee that all the user-file contents are correct after a crash. It ensures only that the file-system data structures (the metadata files) are undamaged and reflect some consistent state that existed prior to the crash. It would be possible to extend the transaction scheme to cover user files, and Microsoft took some steps to do this in Windows Vista.

The log is stored in the third metadata file at the beginning of the volume. It is created with a fixed maximum size when the file system is formatted. It has two sections: the ***logging area***, which is a circular queue of log records, and the ***restart area***, which holds context information, such as the position in the logging area where NTFS should start reading during a recovery. In fact, the restart area holds two copies of its information, so recovery is still possible if one copy is damaged during the crash.

The logging functionality is provided by the ***log-file service***. In addition to writing the log records and performing recovery actions, the log-file service keeps track of the free space in the log file. If the free space gets too low, the log-file service queues pending transactions, and NTFS halts all new I/O operations. After the in-progress operations complete, NTFS calls the cache manager to flush all data and then resets the log file and performs the queued transactions.

### B.5.3 Security

The security of an NTFS volume is derived from the Windows object model. Each NTFS file references a security descriptor, which specifies the owner of the file, and an access-control list, which contains the access permissions granted or denied to each user or group listed. Early versions of NTFS used a separate security descriptor as an attribute of each file. Beginning with Windows 2000, the security-descriptors attribute points to a shared copy, with a significant savings in disk and caching space; many, many files have identical security descriptors.

In normal operation, NTFS does not enforce permissions on traversal of directories in file path names. However, for compatibility with POSIX, these checks can be enabled. Traversal checks are inherently more expensive, since modern parsing of file path names uses prefix matching rather than directory-by-directory parsing of path names. Prefix matching is an algorithm that looks up strings in a cache and finds the entry with the longest match—for example, an entry for `\foo\bar\dir` would be a match for `\foo\bar\dir2\dir3\myfile`. The prefix-matching cache allows path-name traversal to begin much deeper in the tree, saving many steps. Enforcing traversal checks means that the user's access must be checked at each directory level. For instance, a user might lack permission to traverse `\foo\bar`, so starting at the access for `\foo\bar\dir` would be an error.

### B.5.4 Volume Management and Fault Tolerance

`FtDisk` is the fault-tolerant disk driver for Windows. When installed, it provides several ways to combine multiple disk drives into one logical volume so as to improve performance, capacity, or reliability.

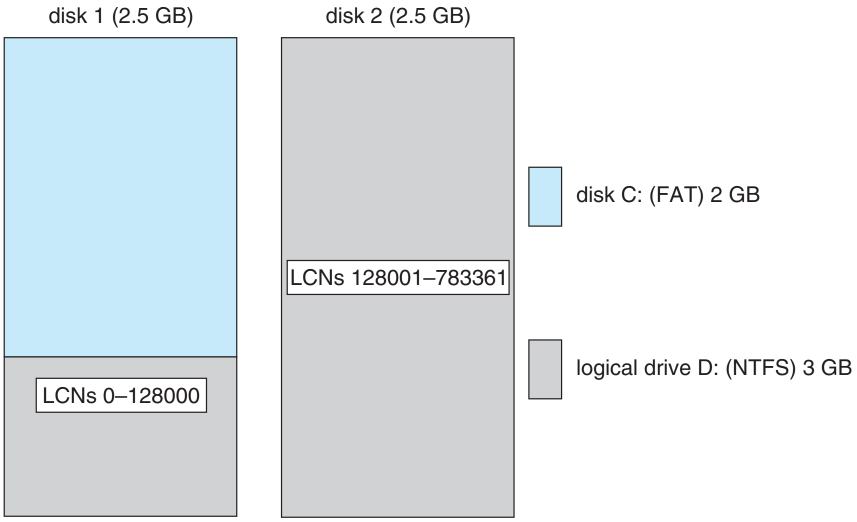


**Figure B.7** Volume set on two drives.

### B.5.4.1 Volume Sets and RAID Sets

One way to combine multiple disks is to concatenate them logically to form a large logical volume, as shown in Figure B.7. In Windows, this logical volume, called a **volume set**, can consist of up to 32 physical partitions. A volume set that contains an NTFS volume can be extended without disturbance of the data already stored in the file system. The bitmap metadata on the NTFS volume are simply extended to cover the newly added space. NTFS continues to use the same LCN mechanism that it uses for a single physical disk, and the `FtDisk` driver supplies the mapping from a logical-volume offset to the offset on one particular disk.

Another way to combine multiple physical partitions is to interleave their blocks in round-robin fashion to form a **stripe set**. This scheme is also called RAID level 0, or **disk striping**. (For more on RAID (redundant arrays of inexpensive disks), see Section 11.8.) `FtDisk` uses a stripe size of 64 KB. The first 64 KB of the logical volume are stored in the first physical partition, the second 64 KB in the second physical partition, and so on, until each partition has contributed 64 KB of space. Then, the allocation wraps around to the first disk, allocating the second 64-KB block. A stripe set forms one large logical volume, but the physical layout can improve the I/O bandwidth, because for a large I/O, all the disks can transfer data in parallel. Windows also supports RAID level 5, stripe set with parity, and RAID level 1, mirroring.

### B.5.4.2 Sector Sparing and Cluster Remapping

To deal with disk sectors that go bad, `FtDisk` uses a hardware technique called sector sparing, and NTFS uses a software technique called cluster remapping. **Sector sparing** is a hardware capability provided by many disk drives. When a disk drive is formatted, it creates a map from logical block numbers to good sectors on the disk. It also leaves extra sectors unmapped, as spares. If a sector fails, `FtDisk` instructs the disk drive to substitute a spare. **Cluster remapping** is a software technique performed by the file system. If a disk block goes bad, NTFS substitutes a different, unallocated block by changing any affected pointers in the MFT. NTFS also makes a note that the bad block should never be allocated to any file.

When a disk block goes bad, the usual outcome is a data loss. But sector sparing or cluster remapping can be combined with fault-tolerant volumes to mask the failure of a disk block. If a read fails, the system reconstructs the missing data by reading the mirror or by calculating the `exclusive or` parity in a stripe set with parity. The reconstructed data are stored in a new location that is obtained by sector sparing or cluster remapping.

### B.5.5 Compression

NTFS can perform data compression on individual files or on all data files in a directory. To compress a file, NTFS divides the file's data into **compression units**, which are blocks of 16 contiguous clusters. When a compression unit is written, a data-compression algorithm is applied. If the result fits into fewer than 16 clusters, the compressed version is stored. When reading, NTFS can determine whether data have been compressed: if they have been, the length of the stored compression unit is less than 16 clusters. To improve performance when reading contiguous compression units, NTFS prefetches and decompresses ahead of the application requests.

For sparse files or files that contain mostly zeros, NTFS uses another technique to save space. Clusters that contain only zeros because they have never been written are not actually allocated or stored on disk. Instead, gaps are left in the sequence of virtual-cluster numbers stored in the MFT entry for the file. When reading a file, if NTFS finds a gap in the virtual-cluster numbers, it just zero-fills that portion of the caller's buffer. This technique is also used by UNIX.

### B.5.6 Mount Points, Symbolic Links, and Hard Links

Mount points are a form of symbolic link specific to directories on NTFS that were introduced in Windows 2000. They provide a mechanism for organizing disk volumes that is more flexible than the use of global names (like drive letters). A mount point is implemented as a symbolic link with associated data that contains the true volume name. Ultimately, mount points will supplant drive letters completely, but there will be a long transition due to the dependence of many applications on the drive-letter scheme.

Windows Vista introduced support for a more general form of symbolic links, similar to those found in UNIX. The links can be absolute or relative, can point to objects that do not exist, and can point to both files and directories even across volumes. NTFS also supports **hard links**, where a single file has an entry in more than one directory of the same volume.

### B.5.7 Change Journal

NTFS keeps a journal describing all changes that have been made to the file system. User-mode services can receive notifications of changes to the journal and then identify what files have changed by reading from the journal. The search indexer service uses the change journal to identify files that need to be re-indexed. The file-replication service uses it to identify files that need to be replicated across the network.

### B.5.8 Volume Shadow Copies

Windows implements the capability of bringing a volume to a known state and then creating a shadow copy that can be used to back up a consistent view of the volume. This technique is known as ***snapshots*** in some other file systems. Making a shadow copy of a volume is a form of copy-on-write, where blocks modified after the shadow copy is created are stored in their original form in the copy. To achieve a consistent state for the volume requires the cooperation of applications, since the system cannot know when the data used by the application are in a stable state from which the application could be safely restarted.

The server version of Windows uses shadow copies to efficiently maintain old versions of files stored on file servers. This allows users to see documents stored on file servers as they existed at earlier points in time. The user can use this feature to recover files that were accidentally deleted or simply to look at a previous version of the file, all without pulling out backup media.

## B.6 Networking

Windows supports both peer-to-peer and client–server networking. It also has facilities for network management. The networking components in Windows provide data transport, interprocess communication, file sharing across a network, and the ability to send print jobs to remote printers.

### B.6.1 Network Interfaces

To describe networking in Windows, we must first mention two of the internal networking interfaces: the **network device interface specification** (**NDIS**) and the **transport driver interface** (**TDI**). The NDIS interface was developed in 1989 by Microsoft and 3Com to separate network adapters from transport protocols so that either could be changed without affecting the other. NDIS resides at the interface between the data-link and network layers in the ISO model and enables many protocols to operate over many different network adapters. In terms of the ISO model, the TDI is the interface between the transport layer (layer 4) and the session layer (layer 5). This interface enables any session-layer component to use any available transport mechanism. (Similar reasoning led to the streams mechanism in UNIX.) The TDI supports both connection-based and connectionless transport and has functions to send any type of data.

### B.6.2 Protocols

Windows implements transport protocols as drivers. These drivers can be loaded and unloaded from the system dynamically, although in practice the system typically has to be rebooted after a change. Windows comes with several networking protocols. Next, we discuss a number of these protocols.

#### B.6.2.1 Server-Message Block

The **server-message-block** (**SMB**) protocol was first introduced in MS-DOS 3.1. The system uses the protocol to send I/O requests over the network. The SMB protocol has four message types. `Session control` messages are commands that start and end a redirector connection to a shared resource at the server. A redirector uses `File` messages to access files at the server. `Printer` messages are used to send data to a remote print queue and to receive status information from the queue, and `Message` messages are used to communicate with another workstation. A version of the SMB protocol was published as the **common Internet file system** (**CIFS**) and is supported on a number of operating systems.

#### B.6.2.2 Transmission Control Protocol/Internet Protocol

The transmission control protocol/Internet protocol (TCP/IP) suite that is used on the Internet has become the de facto standard networking infrastructure. Windows uses TCP/IP to connect to a wide variety of operating systems and hardware platforms. The Windows TCP/IP package includes the simple network-management protocol (SNM), the dynamic host-configuration protocol (DHCP), and the older Windows Internet name service (WINS). Windows Vista introduced a new implementation of TCP/IP that supports both `IPv4` and `IPv6` in the same network stack. This new implementation also supports offloading of the network stack onto advanced hardware, to achieve very high performance for servers.

Windows provides a software firewall that limits the TCP ports that can be used by programs for network communication. Network firewalls are commonly implemented in routers and are a very important security measure. Having a firewall built into the operating system makes a hardware router unnecessary, and it also provides more integrated management and easier use.

#### B.6.2.3 Point-to-Point Tunneling Protocol

The **point-to-point tunneling protocol** (**PPTP**) is a protocol provided by Windows to communicate between remote-access server modules running on Windows server machines and other client systems that are connected over the Internet. The remote-access servers can encrypt data sent over the connection, and they support multiprotocol **virtual private networks** (**VPNs**) over the Internet.

#### B.6.2.4 HTTP Protocol

The HTTP protocol is used to `get/put` information using the World Wide Web. Windows implements HTTP using a kernel-mode driver, so web servers can operate with a low-overhead connection to the networking stack. HTTP is a fairly general protocol that Windows makes available as a transport option for implementing RPC.

#### B.6.2.5 Web-Distributed Authoring and Versioning Protocol

Web-distributed authoring and versioning (WebDAV) is an HTTP-based protocol for collaborative authoring across a network. Windows builds a WebDAV redirector into the file system. Being built directly into the file system enables WebDAV to work with other file-system features, such as encryption. Personal files can then be stored securely in a public place. Because WebDAV uses HTTP, which is a `get/put` protocol, Windows has to cache the files locally so programs can use `read` and `write` operations on parts of the files.

#### B.6.2.6 Named Pipes

**Named pipes** are a connection-oriented messaging mechanism. A process can use named pipes to communicate with other processes on the same machine. Since named pipes are accessed through the file-system interface, the security mechanisms used for file objects also apply to named pipes. The SMB protocol supports named pipes, so they can also be used for communication between processes on different systems.

The format of pipe names follows the **uniform naming convention** (**UNC**). A UNC name looks like a typical remote file name. The format is `\\server_name\share_name\x\y\z`, where `server_name` identifies a server on the network; `share_name` identifies any resource that is made available to network users, such as directories, files, named pipes, and printers; and `\x\y\z` is a normal file path name.

#### B.6.2.7 Remote Procedure Calls

A remote procedure call (RPC) is a client–server mechanism that enables an application on one machine to make a procedure call to code on another machine. The client calls a local procedure—a stub routine—that packs its arguments into a message and sends them across the network to a particular server process. The client-side stub routine then blocks. Meanwhile, the server unpacks the message, calls the procedure, packs the return results into a message, and sends them back to the client stub. The client stub unblocks, receives the message, unpacks the results of the RPC, and returns them to the caller. This packing of arguments is sometimes called **marshaling**. The client stub code and the descriptors necessary to pack and unpack the arguments for an RPC are compiled from a specification written in the **Microsoft Interface Definitio Language**.

The Windows RPC mechanism follows the widely used distributed-computing-environment standard for RPC messages, so programs written to use Windows RPCs are highly portable. The RPC standard is detailed. It hides many of the architectural differences among computers, such as the sizes of binary numbers and the order of bytes and bits in computer words, by specifying standard data formats for RPC messages.

#### B.6.2.8 Component Object Model

The **component object model** (**COM**) is a mechanism for interprocess communication that was developed for Windows. COM objects provide a well-defined interface to manipulate the data in the object. For instance, COM is the infrastructure used by Microsoft's **object linking and embedding** (**OLE**) technology for inserting spreadsheets into Microsoft Word documents. Many Windows services provide COM interfaces. Windows has a distributed extension called

**DCOM** that can be used over a network utilizing RPC to provide a transparent method of developing distributed applications.

### B.6.3 Redirectors and Servers

In Windows, an application can use the Windows I/O API to access files from a remote computer as though they were local, provided that the remote computer is running a CIFS server such as those provided by Windows. A **redirector** is the client-side object that forwards I/O requests to a remote system, where they are satisfied by a server. For performance and security, the redirectors and servers run in kernel mode.

In more detail, access to a remote file occurs as follows:

1. The application calls the I/O manager to request that a file be opened with a file name in the standard UNC format.
2. The I/O manager builds an I/O request packet, as described in Section B.3.3.5.
3. The I/O manager recognizes that the access is for a remote file and calls a driver called a **multiple universal-naming-convention provider** (**MUP**).
4. The MUP sends the I/O request packet asynchronously to all registered redirectors.
5. A redirector that can satisfy the request responds to the MUP. To avoid asking all the redirectors the same question in the future, the MUP uses a cache to remember which redirector can handle this file.
6. The redirector sends the network request to the remote system.
7. The remote-system network drivers receive the request and pass it to the server driver.
8. The server driver hands the request to the proper local file-system driver.
9. The proper device driver is called to access the data.
10. The results are returned to the server driver, which sends the data back to the requesting redirector. The redirector then returns the data to the calling application via the I/O manager.

A similar process occurs for applications that use the Win32 network API, rather than the UNC services, except that a module called a ***multi-provider router*** is used instead of a MUP.

For portability, redirectors and servers use the TDI API for network transport. The requests themselves are expressed in a higher-level protocol, which by default is the SMB protocol described in Section B.6.2. The list of redirectors is maintained in the system hive of the registry.

#### B.6.3.1 Distributed File System

UNC names are not always convenient, because multiple file servers may be available to serve the same content and UNC names explicitly include the name of the server. Windows supports a **distributed file-syste** (**DFS**) protocol that allows a network administrator to serve up files from multiple servers using a single distributed name space.

### B.6.3.2 Folder Redirection and Client-Side Caching

To improve the PC experience for users who frequently switch among computers, Windows allows administrators to give users **roaming profile** , which keep users' preferences and other settings on servers. **Folder redirection** is then used to automatically store a user's documents and other files on a server.

This works well until one of the computers is no longer attached to the network, as when a user takes a laptop onto an airplane. To give users off-line access to their redirected files, Windows uses **client-side caching** (**CSC**). CSC is also used when the computer is on-line to keep copies of the server files on the local machine for better performance. The files are pushed up to the server as they are changed. If the computer becomes disconnected, the files are still available, and the update of the server is deferred until the next time the computer is online.

## B.6.4 Domains

Many networked environments have natural groups of users, such as students in a computer laboratory at school or employees in one department in a business. Frequently, we want all the members of the group to be able to access shared resources on their various computers in the group. To manage the global access rights within such groups, Windows uses the concept of a domain. Previously, these domains had no relationship whatsoever to the domain-name system (DNS) that maps Internet host names to IP addresses. Now, however, they are closely related.

Specifically, a Windows domain is a group of Windows workstations and servers that share a common security policy and user database. Since Windows uses the Kerberos protocol for trust and authentication, a Windows domain is the same thing as a Kerberos realm. Windows uses a hierarchical approach for establishing trust relationships between related domains. The trust relationships are based on DNS and allow transitive trusts that can flow up and down the hierarchy. This approach reduces the number of trusts required for $n$ domains from $n * (n - 1)$ to $O(n)$. The workstations in the domain trust the domain controller to give correct information about the access rights of each user (loaded into the user's access token by LSASS). All users retain the ability to restrict access to their own workstations, however, no matter what any domain controller may say.

## B.6.5 Active Directory

**Active Directory** is the Windows implementation of **lightweight directory-access protocol** (**LDAP**) services. Active Directory stores the topology information about the domain, keeps the domain-based user and group accounts and passwords, and provides a domain-based store for Windows features that need it, such as **Windows group policy**. Administrators use group policies to establish uniform standards for desktop preferences and software. For many corporate information-technology groups, this uniformity drastically reduces the cost of computing.

## B.7 Programmer Interface

The **Win32 API** is the fundamental interface to the capabilities of Windows. This section describes five main aspects of the Win32 API: access to kernel objects, sharing of objects between processes, process management, interprocess communication, and memory management.

### B.7.1 Access to Kernel Objects

The Windows kernel provides many services that application programs can use. Application programs obtain these services by manipulating kernel objects. A process gains access to a kernel object named `XXX` by calling the `CreateXXX` function to open a handle to an instance of `XXX`. This handle is unique to the process. Depending on which object is being opened, if the `Create()` function fails, it may return 0, or it may return a special constant named `INVALID_HANDLE_VALUE`. A process can close any handle by calling the `CloseHandle()` function, and the system may delete the object if the count of handles referencing the object in all processes drops to zero.

### B.7.2 Sharing Objects between Processes

Windows provides three ways to share objects between processes. The first way is for a child process to inherit a handle to the object. When the parent calls the `CreateXXX` function, the parent supplies a `SECURITIES_ATTRIBUTES` structure with the `bInheritHandle` field set to `TRUE`. This field creates an inheritable handle. Next, the child process is created, passing a value of `TRUE` to the `CreateProcess()` function's `bInheritHandle` argument. Figure B.8 shows a code sample that creates a semaphore handle inherited by a child process.

```
SECURITY_ATTRIBUTES sa;
sa.nlength = sizeof(sa);
sa.lpSecurityDescriptor = NULL;
sa.bInheritHandle = TRUE;
Handle a_semaphore = CreateSemaphore(&sa, 1, 1, NULL);
char comand_line[132];
ostrstream ostring(command_line, sizeof(command_line));
ostring << a_semaphore << ends;
CreateProcess("another_process.exe", command_line,
   NULL, NULL, TRUE, . . .);
```

**Figure B.8** Code enabling a child to share an object by inheriting a handle.

```
// Process A
. . .
HANDLE a_semaphore = CreateSemaphore(NULL, 1, 1, "MySEM1");
. . .

// Process B
. . .
HANDLE b_semaphore = OpenSemaphore(SEMAPHORE_ALL_ACCESS,
   FALSE, "MySEM1");
. . .
```

**Figure B.9** Code for sharing an object by name lookup.

Assuming the child process knows which handles are shared, the parent and child can achieve interprocess communication through the shared objects. In the example in Figure B.8, the child process gets the value of the handle from the first command-line argument and then shares the semaphore with the parent process.

The second way to share objects is for one process to give the object a name when the object is created and for the second process to open the name. This method has two drawbacks: Windows does not provide a way to check whether an object with the chosen name already exists, and the object name space is global, without regard to the object type. For instance, two applications may create and share a single object named "foo" when two distinct objects—possibly of different types—were desired.

Named objects have the advantage that unrelated processes can readily share them. The first process calls one of the `CreateXXX` functions and supplies a name as a parameter. The second process gets a handle to share the object by calling `OpenXXX()` (or `CreateXXX`) with the same name, as shown in the example in Figure B.9.

The third way to share objects is via the `DuplicateHandle()` function. This method requires some other method of interprocess communication to pass the duplicated handle. Given a handle to a process and the value of a handle within that process, a second process can get a handle to the same object and thus share it. An example of this method is shown in Figure B.10.

### B.7.3 Process Management

In Windows, a **process** is a loaded instance of an application and a **thread** is an executable unit of code that can be scheduled by the kernel dispatcher. Thus, a process contains one or more threads. A process is created when a thread in some other process calls the `CreateProcess()` API. This routine loads any dynamic link libraries used by the process and creates an initial thread in the process. Additional threads can be created by the `CreateThread()` function. Each thread is created with its own stack, which defaults to 1 MB unless otherwise specified in an argument to `CreateThread()`.

```
// Process A wants to give Process B access to a semaphore

// Process A
HANDLE a_semaphore = CreateSemaphore(NULL, 1, 1, NULL);
// send the value of the semaphore to Process B
// using a message or shared memory object
. . .

// Process B
HANDLE process_a = OpenProcess(PROCESS_ALL_ACCESS, FALSE,
   process_id_of_A);
HANDLE b_semaphore;
DuplicateHandle(process_a, a_semaphore,
   GetCurrentProcess(), &b_semaphore,
   0, FALSE, DUPLICATE_SAME_ACCESS);
// use b_semaphore to access the semaphore
. . .
```

**Figure B.10** Code for sharing an object by passing a handle.

#### B.7.3.1 Scheduling Rule

Priorities in the Win32 environment are based on the native kernel (NT) scheduling model, but not all priority values may be chosen. The Win32 API uses four priority classes:

1. `IDLE_PRIORITY_CLASS` (NT priority level 4)
2. `NORMAL_PRIORITY_CLASS` (NT priority level 8)
3. `HIGH_PRIORITY_CLASS` (NT priority level 13)
4. `REALTIME_PRIORITY_CLASS` (NT priority level 24)

Processes are typically members of the `NORMAL_PRIORITY_CLASS` unless the parent of the process was of the `IDLE_PRIORITY_CLASS` or another class was specified when `CreateProcess` was called. The priority class of a process is the default for all threads that execute in the process. It can be changed with the `SetPriorityClass()` function or by passing an argument to the `START` command. Only users with the ***increase scheduling priority*** privilege can move a process into the `REALTIME_PRIORITY_CLASS`. Administrators and power users have this privilege by default.

When a user is running an interactive process, the system needs to schedule the process's threads to provide good responsiveness. For this reason, Windows has a special scheduling rule for processes in the `NORMAL_PRIORITY_CLASS`. Windows distinguishes between the process associated with the foreground window on the screen and the other (background) processes. When a process moves into the foreground, Windows increases the scheduling quantum for all its threads by a factor of 3; CPU-bound threads in the foreground process will run three times longer than similar threads in background processes.

### B.7.3.2 Thread Priorities

A thread starts with an initial priority determined by its class. The priority can be altered by the `SetThreadPriority()` function. This function takes an argument that specifies a priority relative to the base priority of its class:

- `THREAD_PRIORITY_LOWEST`: base − 2
- `THREAD_PRIORITY_BELOW_NORMAL`: base − 1
- `THREAD_PRIORITY_NORMAL`: base + 0
- `THREAD_PRIORITY_ABOVE_NORMAL`: base + 1
- `THREAD_PRIORITY_HIGHEST`: base + 2

Two other designations are also used to adjust the priority. Recall from Section B.3.2.2 that the kernel has two priority classes: 16–31 for the real-time class and 1–15 for the variable class. `THREAD_PRIORITY_IDLE` sets the priority to 16 for real-time threads and to 1 for variable-priority threads. `THREAD_PRIORITY_TIME_CRITICAL` sets the priority to 31 for real-time threads and to 15 for variable-priority threads.

As discussed in Section B.3.2.2, the kernel adjusts the priority of a variable class thread dynamically depending on whether the thread is I/O bound or CPU bound. The Win32 API provides a method to disable this adjustment via `SetProcessPriorityBoost()` and `SetThreadPriorityBoost()` functions.

### B.7.3.3 Thread Suspend and Resume

A thread can be created in a ***suspended state*** or can be placed in a suspended state later by use of the `SuspendThread()` function. Before a suspended thread can be scheduled by the kernel dispatcher, it must be moved out of the suspended state by use of the `ResumeThread()` function. Both functions set a counter so that if a thread is suspended twice, it must be resumed twice before it can run.

### B.7.3.4 Thread Synchronization

To synchronize concurrent access to shared objects by threads, the kernel provides synchronization objects, such as semaphores and mutexes. These are dispatcher objects, as discussed in Section B.3.2.2. Threads can also synchronize with kernel services operating on kernel objects—such as threads, processes, and files—because these are also dispatcher objects. Synchronization with kernel dispatcher objects can be achieved by use of the `WaitForSingleObject()` and `WaitForMultipleObjects()` functions; these functions wait for one or more dispatcher objects to be signaled.

Another method of synchronization is available to threads within the same process that want to execute code exclusively. The Win32 **critical section object** is a user-mode mutex object that can often be acquired and released without entering the kernel. On a multiprocessor, a Win32 critical section will attempt to spin while waiting for a critical section held by another thread to be released.

If the spinning takes too long, the acquiring thread will allocate a kernel mutex and yield its CPU. Critical sections are particularly efficient because the kernel mutex is allocated only when there is contention and then used only after attempting to spin. Most mutexes in programs are never actually contended, so the savings are significant.

Before using a critical section, some thread in the process must call `InitializeCriticalSection()`. Each thread that wants to acquire the mutex calls `EnterCriticalSection()` and then later calls `LeaveCriticalSection()` to release the mutex. There is also a `TryEnterCriticalSection()` function, which attempts to acquire the mutex without blocking.

For programs that want user-mode reader–writer locks rather than a mutex, Win32 supports **slim reader–writer** (**SRW**) **locks**. SRW locks have APIs similar to those for critical sections, such as `InitializeSRWLock`, `AcquireSRWLockXXX`, and `ReleaseSRWLockXXX`, where `XXX` is either `Exclusive` or `Shared`, depending on whether the thread wants write access or just read access to the object protected by the lock. The Win32 API also supports **condition variables**, which can be used with either critical sections or SRW locks.

#### B.7.3.5 Thread Pool

Repeatedly creating and deleting threads can be expensive for applications and services that perform small amounts of work in each instantiation. The Win32 thread pool provides user-mode programs with three services: a queue to which work requests may be submitted (via the `SubmitThreadpoolWork()` function), an API that can be used to bind callbacks to waitable handles (`RegisterWaitForSingleObject()`), and APIs to work with timers (`CreateThreadpoolTimer()` and `WaitForThreadpoolTimerCallbacks()`) and to bind callbacks to I/O completion queues (`BindIoCompletionCallback()`).

The goal of using a thread pool is to increase performance and reduce memory footprint. Threads are relatively expensive, and each processor can only be executing one thread at a time no matter how many threads are available. The thread pool attempts to reduce the number of runnable threads by slightly delaying work requests (reusing each thread for many requests) while providing enough threads to effectively utilize the machine's CPUs. The wait and I/O- and timer-callback APIs allow the thread pool to further reduce the number of threads in a process, using far fewer threads than would be necessary if a process were to devote separate threads to servicing each waitable handle, timer, or completion port.

#### B.7.3.6 Fibers

A **fibe** is user-mode code that is scheduled according to a user-defined scheduling algorithm. Fibers are completely a user-mode facility; the kernel is not aware that they exist. The fiber mechanism uses Windows threads as if they were CPUs to execute the fibers. Fibers are cooperatively scheduled, meaning that they are never preempted but must explicitly yield the thread on which they are running. When a fiber yields a thread, another fiber can be scheduled on it by the run-time system (the programming language run-time code).

The system creates a fiber by calling either `ConvertThreadToFiber()` or `CreateFiber()`. The primary difference between these functions is that

`CreateFiber()` does not begin executing the fiber that was created. To begin execution, the application must call `SwitchToFiber()`. The application can terminate a fiber by calling `DeleteFiber()`.

Fibers are not recommended for threads that use Win32 APIs rather than standard C-library functions because of potential incompatibilities. Win32 user-mode threads have a **thread-environment block** (**TEB**) that contains numerous per-thread fields used by the Win32 APIs. Fibers must share the TEB of the thread on which they are running. This can lead to problems when a Win32 interface puts state information into the TEB for one fiber and then the information is overwritten by a different fiber. Fibers are included in the Win32 API to facilitate the porting of legacy UNIX applications that were written for a user-mode thread model such as Pthreads.

#### B.7.3.7 User-Mode Scheduling (UMS) and ConcRT

A new mechanism in Windows 7, user-mode scheduling (UMS), addresses several limitations of fibers. First, recall that fibers are unreliable for executing Win32 APIs because they do not have their own TEBs. When a thread running a fiber blocks in the kernel, the user scheduler loses control of the CPU for a time as the kernel dispatcher takes over scheduling. Problems may result when fibers change the kernel state of a thread, such as the priority or impersonation token, or when they start asynchronous I/O.

UMS provides an alternative model by recognizing that each Windows thread is actually two threads: a kernel thread (KT) and a user thread (UT). Each type of thread has its own stack and its own set of saved registers. The KT and UT appear as a single thread to the programmer because UTs can never block but must always enter the kernel, where an implicit switch to the corresponding KT takes place. UMS uses each UT's TEB to uniquely identify the UT. When a UT enters the kernel, an explicit switch is made to the KT that corresponds to the UT identified by the current TEB. The reason the kernel does not know which UT is running is that UTs can invoke a user-mode scheduler, as fibers do. But in UMS, the scheduler switches UTs, including switching the TEBs.

When a UT enters the kernel, its KT may block. When this happens, the kernel switches to a scheduling thread, which UMS calls a ***primary***, and uses this thread to reenter the user-mode scheduler so that it can pick another UT to run. Eventually, a blocked KT will complete its operation and be ready to return to user mode. Since UMS has already reentered the user-mode scheduler to run a different UT, UMS queues the UT corresponding to the completed KT to a completion list in user mode. When the user-mode scheduler is choosing a new UT to switch to, it can examine the completion list and treat any UT on the list as a candidate for scheduling.

Unlike fibers, UMS is not intended to be used directly by the programmer. The details of writing user-mode schedulers can be very challenging, and UMS does not include such a scheduler. Rather, the schedulers come from programming language libraries that build on top of UMS. Microsoft Visual Studio 2010 shipped with Concurrency Runtime (ConcRT), a concurrent programming framework for C++. ConcRT provides a user-mode scheduler together with facilities for decomposing programs into tasks, which can then be scheduled on the available CPUs. ConcRT provides support for `par_for` styles of con-

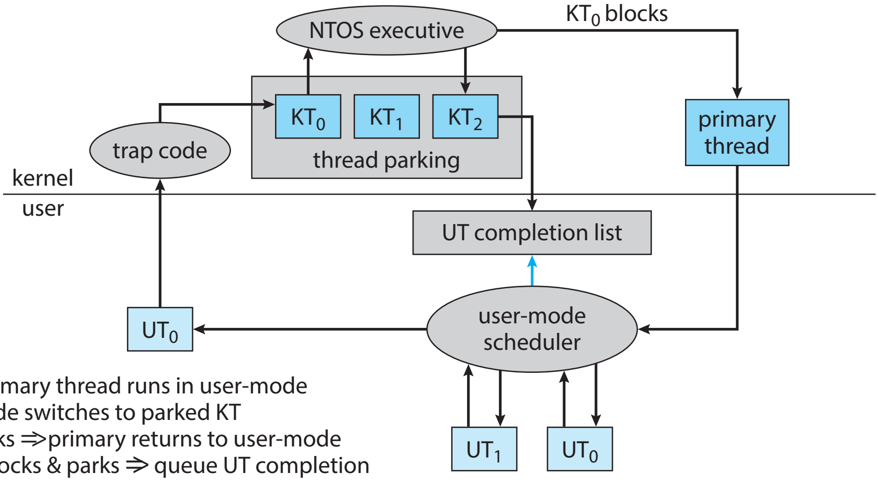

NTOS executive
KT0 blocks
KT0
KT1
KT2
thread parking
trap code
primary thread
kernel
user
UT completion list
user-mode scheduler
UT0
UT1
UT0
Only primary thread runs in user-mode
Trap code switches to parked KT
KT blocks ⇒primary returns to user-mode
KT unblocks & parks ⇒ queue UT completion

**Figure B.11** User-mode scheduling.

structs, as well as rudimentary resource management and task synchronization primitives. The key features of UMS are depicted in Figure B.11.

#### B.7.3.8 Winsock

**Winsock** is the Windows sockets API. Winsock is a session-layer interface that is largely compatible with UNIX sockets but has some added Windows extensions. It provides a standardized interface to many transport protocols that may have different addressing schemes, so that any Winsock application can run on any Winsock-compliant protocol stack. Winsock underwent a major update in Windows Vista to add tracing, `IPv6` support, impersonation, new security APIs and many other features.

Winsock follows the Windows Open System Architecture (WOSA) model, which provides a standard service provider interface (SPI) between applications and networking protocols. Applications can load and unload ***layered protocols*** that build additional functionality, such as additional security, on top of the transport protocol layers. Winsock supports asynchronous operations and notifications, reliable multicasting, secure sockets, and kernel mode sockets. There is also support for simpler usage models, like the `WSAConnectByName()` function, which accepts the target as strings specifying the name or IP address of the server and the service or port number of the destination port.

### B.7.4 IPC Using Windows Messaging

Win32 applications handle interprocess communication in several ways. One way is by using shared kernel objects. Another is by using the Windows messaging facility, an approach that is particularly popular for Win32 GUI applications. One thread can send a message to another thread or to a window by calling `PostMessage()`, `PostThreadMessage()`, `SendMessage()`, `SendThreadMessage()`, or `SendMessageCallback()`. ***Posting*** a message and ***sending*** a message differ in this way: the post routines are asynchronous, they return immediately, and the calling thread does not know when the message

```
// allocate 16 MB at the top of our address space
void *buf = VirtualAlloc(0, 0x1000000, MEM_RESERVE | MEM_TOP_DOWN,
   PAGE_READWRITE);
// commit the upper 8 MB of the allocated space
VirtualAlloc(buf + 0x800000, 0x800000, MEM_COMMIT, PAGE_READWRITE);
// do something with the memory
. . .
// now decommit the memory
VirtualFree(buf + 0x800000, 0x800000, MEM_DECOMMIT);
// release all of the allocated address space
VirtualFree(buf, 0, MEM_RELEASE);
```

**Figure B.12** Code fragments for allocating virtual memory.

is actually delivered. The send routines are synchronous: they block the caller until the message has been delivered and processed.

In addition to sending a message, a thread can send data with the message. Since processes have separate address spaces, the data must be copied. The system copies data by calling `SendMessage()` to send a message of type `WM_COPYDATA` with a `COPYDATASTRUCT` data structure that contains the length and address of the data to be transferred. When the message is sent, Windows copies the data to a new block of memory and gives the virtual address of the new block to the receiving process.

Every Win32 thread has its own input queue from which it receives messages. If a Win32 application does not call `GetMessage()` to handle events on its input queue, the queue fills up, and after about five seconds, the system marks the application as “Not Responding”.

## B.7.5 Memory Management

The Win32 API provides several ways for an application to use memory: virtual memory, memory-mapped files, heaps, and thread-local storage.

### B.7.5.1 Virtual Memory

An application calls `VirtualAlloc()` to reserve or commit virtual memory and `VirtualFree()` to decommit or release the memory. These functions enable the application to specify the virtual address at which the memory is allocated. They operate on multiples of the memory page size. Examples of these functions appear in Figure B.12.

A process may lock some of its committed pages into physical memory by calling `VirtualLock()`. The maximum number of pages a process can lock is 30, unless the process first calls `SetProcessWorkingSetSize()` to increase the maximum working-set size.

### B.7.5.2 Memory-Mapping Files

Another way for an application to use memory is by memory-mapping a file into its address space. Memory mapping is also a convenient way for two

```
// open the file or create it if it does not exist
HANDLE hfile = CreateFile("somefile", GENERIC_READ | GENERIC_WRITE,
   FILE_SHARE_READ | FILE_SHARE_WRITE, NULL,
   OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
// create the file mapping 8 MB in size
HANDLE hmap = CreateFileMapping(hfile, PAGE_READWRITE,
   SEC_COMMIT, 0, 0x800000, "SHM_1");
// now get a view of the space mapped
void *buf = MapViewOfFile(hmap, FILE_MAP_ALL_ACCESS,
   0, 0, 0, 0x800000);
// do something with the mapped file
. . .
// now unmap the file
UnMapViewOfFile(buf);
CloseHandle(hmap);
CloseHandle(hfile);
```

**Figure B.13** Code fragments for memory mapping of a file.

processes to share memory: both processes map the same file into their virtual memory. Memory mapping is a multistage process, as you can see in the example in Figure B.13.

If a process wants to map some address space just to share a memory region with another process, no file is needed. The process calls `CreateFileMapping()` with a file handle of `0xffffffff` and a particular size. The resulting file-mapping object can be shared by inheritance, by name lookup, or by handle duplication.

#### B.7.5.3 Heaps

Heaps provide a third way for applications to use memory, just as with `malloc()` and `free()` in standard C. A heap in the Win32 environment is a region of reserved address space. When a Win32 process is initialized, it is created with a **default heap**. Since most Win32 applications are multithreaded, access to the heap is synchronized to protect the heap's space-allocation data structures from being damaged by concurrent updates by multiple threads.

Win32 provides several heap-management functions so that a process can allocate and manage a private heap. These functions are `HeapCreate()`, `HeapAlloc()`, `HeapRealloc()`, `HeapSize()`, `HeapFree()`, and `HeapDestroy()`. The Win32 API also provides the `HeapLock()` and `HeapUnlock()` functions to enable a thread to gain exclusive access to a heap. Unlike `VirtualLock()`, these functions perform only synchronization; they do not lock pages into physical memory.

The original Win32 heap was optimized for efficient use of space. This led to significant problems with fragmentation of the address space for larger server programs that ran for long periods of time. A new **low-fragmentation heap** (**LFH**) design introduced in Windows XP greatly reduced the fragmen-

```
// reserve a slot for a variable
DWORD var_index = T1sAlloc();
// set it to the value 10
T1sSetValue(var_index, 10);
// get the value
int var T1sGetValue(var_index);
// release the index
T1sFree(var_index);
```

**Figure B.14** Code for dynamic thread-local storage.

tation problem. The Windows 7 heap manager automatically turns on LFH as appropriate.

### B.7.5.4 Thread-Local Storage

A fourth way for applications to use memory is through a **thread-local storage** (**TLS**) mechanism. Functions that rely on global or static data typically fail to work properly in a multithreaded environment. For instance, the C run-time function `strtok()` uses a static variable to keep track of its current position while parsing a string. For two concurrent threads to execute `strtok()` correctly, they need separate `current position` variables. TLS provides a way to maintain instances of variables that are global to the function being executed but not shared with any other thread.

TLS provides both dynamic and static methods of creating thread-local storage. The dynamic method is illustrated in Figure B.14. The TLS mechanism allocates global heap storage and attaches it to the thread environment block that Windows allocates to every user-mode thread. The TEB is readily accessible by each thread and is used not just for TLS but for all the per-thread state information in user mode.

To use a thread-local static variable, the application declares the variable as follows to ensure that every thread has its own private copy:

```
__declspec(thread) DWORD cur_pos = 0;
```

## B.8 Summary

Microsoft designed Windows to be an extensible, portable operating system—one able to take advantage of new techniques and hardware. Windows supports multiple operating environments and symmetric multiprocessing, including both 32-bit and 64-bit processors and NUMA computers. The use of kernel objects to provide basic services, along with support for client–server computing, enables Windows to support a wide variety of application environments. Windows provides virtual memory, integrated caching, and preemptive scheduling. It supports elaborate security mechanisms and includes internationalization features. Windows runs on a wide variety of computers, so users can choose and upgrade hardware to match their budgets and performance requirements without needing to alter the applications they run.

## Practice Exercises

**B.1** What type of operating system is Windows? Describe two of its major features.

**B.2** List the design goals of Windows. Describe two in detail.

**B.3** Describe the booting process for a Windows system.

**B.4** Describe the three main architectural layers of the Windows kernel.

**B.5** What is the job of the object manager?

**B.6** What types of services does the process manager provide?

**B.7** What is a local procedure call?

**B.8** What are the responsibilities of the I/O manager?

**B.9** What types of networking does Windows support? How does Windows implement transport protocols? Describe two networking protocols.

**B.10** How is the NTFS namespace organized?

**B.11** How does NTFS handle data structures? How does NTFS recover from a system crash? What is guaranteed after a recovery takes place?

**B.12** How does Windows allocate user memory?

**B.13** Describe some of the ways in which an application can use memory via the Win32 API.

## Further Reading

[Russinovich et al. (2017)] provides an overview of Windows 7 and considerable technical detail about system internals and components. [Brown (2000)] presents details of the security architecture of Windows.

The Microsoft Developer Network Library (http://msdn.microsoft.com) supplies a wealth of information on Windows and other Microsoft products, including documentation of all the published APIs.

[Iseminger (2000)] provides a good reference on the Windows Active Directory. Detailed discussions of writing programs that use the Win32 API appear in [Richter (1997)].

The source code for a 2005 WRK version of the Windows kernel, together with a collection of slides and other CRK curriculum materials, is available from www.microsoft.com/WindowsAcademic for use by universities.

## Bibliography

**[Brown (2000)]** K. Brown, *Programming Windows Security*, Addison-Wesley (2000).

**[Iseminger (2000)]** D. Iseminger, *Active Directory Services for Microsoft Windows 2000. Technical Reference*, Microsoft Press (2000).

**[Richter (1997)]** J. Richter, *Advanced Windows*, Microsoft Press (1997).

**[Russinovich et al. (2017)]** M. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals–Part 1*, Seventh Edition, Microsoft Press (2017).

# Appendix C

# BSD UNIX

**This chapter was first written in 1991 and has been updated over time.**

In Chapter 20, we presented an in-depth examination of the Linux operating system. In this chapter, we examine another popular UNIX version—UnixBSD. We start by presenting a brief history of the UNIX operating system. We then describe the system's user and programmer interfaces. Finally, we discuss the internal data structures and algorithms used by the FreeBSD kernel to support the user–programmer interface.

## C.1 UNIX History

The first version of UNIX was developed in 1969 by Ken Thompson of the Research Group at Bell Laboratories to use an otherwise idle PDP-7. Thompson was soon joined by Dennis Ritchie and they, with other members of the Research Group, produced the early versions of UNIX.

Ritchie had previously worked on the MULTICS project, and MULTICS had a strong influence on the newer operating system. Even the name ***UNIX*** is a pun on ***MULTICS***. The basic organization of the file system, the idea of the command interpreter (or the shell) as a user process, the use of a separate process for each command, the original line-editing characters (# to erase the last character and @ to erase the entire line), and numerous other features came directly from MULTICS. Ideas from other operating systems, such as MIT's CTSS and the XDS-940 system, were also used.

Ritchie and Thompson worked quietly on UNIX for many years. They moved it to a PDP-11/20 for a second version; for a third version, they rewrote most of the operating system in the systems-programming language C, instead of the previously used assembly language. C was developed at Bell Laboratories to support UNIX. UNIX was also moved to larger PDP-11 models, such as the 11/45 and 11/70. Multiprogramming and other enhancements were added when it was rewritten in C and moved to systems (such as the 11/45) that had hardware support for multiprogramming.

As UNIX developed, it became widely used within Bell Laboratories and gradually spread to a few universities. The first version widely available outside Bell Laboratories was Version 6, released in 1976. (The version number for early UNIX systems corresponds to the edition number of the *UNIX Programmer's Manual* that was current when the distribution was made; the code and the manual were revised independently.)

In 1978, Version 7 was distributed. This UNIX system ran on the PDP-11/70 and the Interdata 8/32 and is the ancestor of most modern UNIX systems. In particular, it was soon ported to other PDP-11 models and to the VAX computer line. The version available on the VAX was known as 32V. Research has continued since then.

### C.1.1 UNIX Support Group

After the distribution of Version 7 in 1978, the UNIX Support Group (USG) assumed administrative control and responsibility from the Research Group for distributions of UNIX within AT&T, the parent organization for Bell Laboratories. UNIX was becoming a product, rather than simply a research tool. The Research Group continued to develop their own versions of UNIX, however, to support their internal computing. Version 8 included a facility called the **stream I/O system**, which allows flexible configuration of kernel IPC modules. It also contained RFS, a remote file system similar to Sun's NFS. The current version is Version 10, released in 1989 and available only within Bell Laboratories.

USG mainly provided support for UNIX within AT&T. The first external distribution from USG was System III, in 1982. System III incorporated features of Version 7 and 32V, as well as features of several UNIX systems developed by groups other than Research. For example, features of UNIX/RT, a real-time UNIX system, and numerous portions of the Programmer's Work Bench (PWB) software tools package were included in System III.

USG released System V in 1983; it is largely derived from System III. The divestiture of the various Bell operating companies from AT&T left AT&T in a position to market System V aggressively. USG was restructured as the UNIX System Development Laboratory (USDL), which released UNIX System V Release 2 (V.2) in 1984. UNIX System V Release 2, Version 4 (V.2.4) added a new implementation of virtual memory with copy-on-write paging and shared memory. USDL was in turn replaced by AT&T Information Systems (ATTIS), which distributed System V Release 3 (V.3) in 1987. V.3 adapts the V8 implementation of the stream I/O system and makes it available as ***STREAMS***. It also includes RFS, the NFS-like remote file system mentioned earlier.

### C.1.2 Berkeley Begins Development

The small size, modularity, and clean design of early UNIX systems led to UNIX-based work at numerous other computer-science organizations, such as RAND, BBN, the University of Illinois, Harvard, Purdue, and DEC. The most influential UNIX development group outside of Bell Laboratories and AT&T, however, has been the University of California at Berkeley.

Bill Joy and Ozalp Babaoglu did the first Berkeley VAX UNIX work in 1978. They added virtual memory, demand paging, and page replacement to 32V to produce 3BSD UNIX. This version was the first to implement any of these facilities on a UNIX system. The large virtual memory space of 3BSD allowed the development of very large programs, such as Berkeley's own Franz LISP. The memory-management work convinced the Defense Advanced Research

Projects Agency (DARPA) to fund Berkeley for the development of a standard UNIX system for government use; 4BSD UNIX was the result.

The 4 BSD work for DARPA was guided by a steering committee that included many notable people from the UNIX and networking communities. One of the goals of this project was to provide support for the DARPA Internet networking protocols (TCP/IP). This support was provided in a general manner. It is possible in 4.2 BSD to communicate uniformly among diverse network facilities, including local-area networks (such as Ethernets and token rings) and wide-area networks (such as NSFNET). This implementation was the most important reason for the current popularity of these protocols. Many vendors of UNIX computer systems used it as the basis for their implementations, and it was even used in other operating systems. It permitted the Internet to grow from 60 connected networks in 1984 to more than 8,000 networks and an estimated 10 million users in 1993.

In addition, Berkeley adapted many features from contemporary operating systems to improve the design and implementation of UNIX. Many of the terminal line-editing functions of the TENEX (TOPS-20) operating system were provided by a new terminal driver. A new user interface (the C Shell), a new text editor (ex/vi), compilers for Pascal and LISP, and many new systems programs were written at Berkeley. For 4.2 BSD, certain efficiency improvements were inspired by the VMS operating system.

UNIX software from Berkeley was released in **Berkeley Software Distributions (BSD)**. It is convenient to refer to the Berkeley VAX UNIX systems following 3 BSD as 4 BSD, but there were actually several specific releases, most notably 4.1 BSD and 4.2 BSD; 4.2 BSD, first distributed in 1983, was the culmination of the original Berkeley DARPA UNIX project. The equivalent version for PDP-11 systems was 2.9 BSD.

In 1986, 4.3 BSD was released. It was very similar to 4.2 BSD but included numerous internal changes, such as bug fixes and performance improvements. Some new facilities were also added, including support for the Xerox Network System protocols.

The next version was 4.3 BSD Tahoe, released in 1988. It included improved networking congestion control and TCP/IP performance. Disk configurations were separated from the device drivers and read off the disks themselves. Expanded time-zone support was also included. 4.3 BSD Tahoe was actually developed on and for the CCI Tahoe system (Computer Console, Inc., Power 6 computer), rather than for the usual VAX base. The corresponding PDP-11 release was 2.10.1BSD; it was distributed by the USENIX association, which also published the 4.3 BSD manuals. The 4.3.2 BSD Reno release saw the inclusion of an implementation of ISO/OSI networking.

The last Berkeley release, 4.4 BSD, was finalized in June of 1993. It included new X.25 networking support and POSIX standard compliance. It also had a radically new file system organization, with a new virtual file system interface and support for ***stackable*** file systems, allowing file systems to be layered on top of each other for easy inclusion of new features. An implementation of NFS was included in the release (Section 15.8), along with a new log-based file system (see Chapter 11). The 4.4 BSD virtual memory system was derived from Mach (described in Section A.13). Several other changes, such as enhanced security and improved kernel structure, were also included. With the release of version 4.4, Berkeley halted its research efforts.

### C.1.3 The Spread of UNIX

UNIX 4 BSD was the operating system of choice for the VAX from its initial release (in 1979) until the release of Ultrix, DEC's BSD implementation. Indeed, 4 BSD is still the best choice for many research and networking installations. The current set of UNIX operating systems is not limited to those from Bell Laboratories (which is currently owned by Lucent Technology) and Berkeley, however. Sun Microsystems helped popularize the BSD flavor of UNIX by shipping it on Sun workstations. As UNIX grew in popularity, it was moved to many computers and computer systems. A wide variety of UNIX and UNIX-like operating systems have been created. DEC supported its UNIX (Ultrix) on its workstations and is replacing Ultrix with another UNIX-derived operating system, OSF/1. Microsoft rewrote UNIX for the Intel 8088 family and called it XENIX, and its Windows NT operating system was heavily influenced by UNIX. IBM has UNIX (AIX) on its PCs, workstations, and mainframes. In fact, UNIX is available on almost all general-purpose computers. It runs on personal computers, workstations, minicomputers, mainframes, and supercomputers, from Apple Macintosh IIs to Cray IIs. Because of its wide availability, it is used in environments ranging from academic to military to manufacturing process control. Most of these systems are based on Version 7, System III, 4.2 BSD, or System V.

The wide popularity of UNIX with computer vendors has made UNIX the most portable of operating systems, and users can expect a UNIX environment independent of any specific computer manufacturer. But the large number of implementations of the system has led to remarkable variation in the programming and user interfaces distributed by the vendors. For true vendor independence, application-program developers need consistent interfaces. Such interfaces would allow all "UNIX" applications to run on all UNIX systems, which is certainly not the current situation. This issue has become important as UNIX has become the preferred program-development platform for applications ranging from databases to graphics and networking, and it has led to a strong market demand for UNIX standards.

Several standardization projects have been undertaken. The first was the ***/usr/group 1984 Standard***, sponsored by the UniForum industry user's group. Since then, many official standards bodies have continued the effort, including IEEE and ISO (the POSIX standard). The X/Open Group international consortium completed XPG3, a Common Application Environment, which subsumes the IEEE interface standard. Unfortunately, XPG3 is based on a draft of the ANSI C standard rather than the final specification, and therefore needed to be redone as XPG4. In 1989, the ANSI standards body standardized the C programming language, producing an ANSI C specification that vendors were quick to adopt.

As such projects continue, the flavors of UNIX will converge and lead to one programming interface to UNIX, allowing UNIX to become even more popular. In fact, two separate sets of powerful UNIX vendors are working on this problem: The AT&T-guided UNIX International (UI) and the Open Software Foundation (OSF) have both agreed to follow the POSIX standard. Recently, many of the vendors involved in those two groups have agreed on further standardization (the COSE agreement).

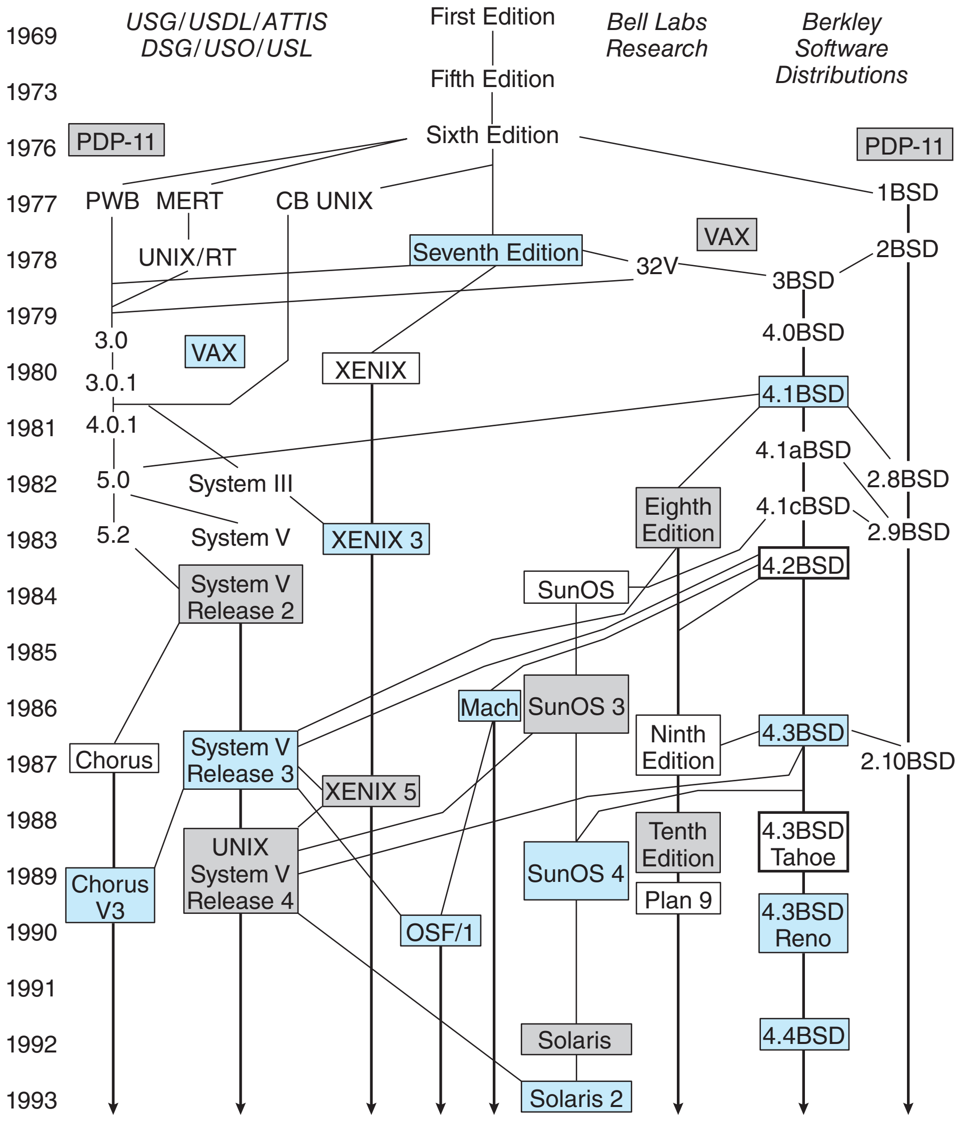

1969
1973
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
USG/USDL/ATTIS
DSG/USO/USL
First Edition
Fifth Edition
Sixth Edition
Bell Labs
Research
Berkley
Software
Distributions
PDP-11
PDP-11
PWB
MERT
CB UNIX
1BSD
UNIX/RT
Seventh Edition
VAX
32V
2BSD
3BSD
3.0
VAX
4.0BSD
3.0.1
XENIX
4.1BSD
4.0.1
4.1aBSD
5.0
System III
2.8BSD
Eighth
Edition
4.1cBSD
5.2
System V
XENIX 3
2.9BSD
4.2BSD
System V
Release 2
SunOS
Mach
SunOS 3
Ninth
Edition
4.3BSD
Chorus
System V
Release 3
2.10BSD
XENIX 5
UNIX
System V
Release 4
Tenth
Edition
4.3BSD
Tahoe
Chorus
V3
SunOS 4
Plan 9
OSF/1
4.3BSD
Reno
Solaris
4.4BSD
Solaris 2

**Figure C.1** History of UNIX versions up to 1993.

AT&T replaced its ATTIS group in 1989 with the UNIX Software Organization (USO), which shipped the first merged UNIX, System V Release 4. This system combines features from System V, 4.3 BSD, and Sun's SunOS, including long file names, the Berkeley file system, virtual memory management, symbolic links, multiple access groups, job control, and reliable signals; it also conforms to the published POSIX standard, POSIX.1. After USO produced SVR4, it became an independent AT&T subsidiary named Unix System Laboratories (USL); in 1993, it was purchased by Novell, Inc. Figure C.1 summarizes the relationships among the various versions of UNIX.

The UNIX system has grown from a personal project of two Bell Laboratories employees to an operating system defined by multinational standardization bodies. At the same time, UNIX is an excellent vehicle for academic study, and we believe it will remain an important part of operating-system theory and practice. For example, the Tunis operating system, the Xinu operating system, and the Minix operating system are based on the concepts of UNIX but were developed explicitly for classroom study. There is a plethora of ongoing UNIX-related research systems, including Mach, Chorus, Comandos, and Roisin. The original developers, Ritchie and Thompson, were honored in 1983 by the Association for Computing Machinery Turing Award for their work on UNIX.

### C.1.4 History of FreeBSD

The specific UNIX version used in this chapter is the Intel version of FreeBSD. This system implements many interesting operating-system concepts, such as demand paging with clustering, as well as networking. The FreeBSD project began in early 1993 to produce a snapshot of 386 BSD to solve problems that could not be resolved using the existing patch mechanism. 386 BSD was derived from 4.3 BSD-Lite (Net/2) and was released in June 1992 by William Jolitz. FreeBSD (coined by David Greenman) 1.0 was released in December 1993, and FreeBSD 1.1 was released in May 1994. Both versions were based on 4.3 BSD-Lite. Legal issues between UCB and Novell required that 4.3 BSD-Lite code no longer be used, so the final 4.3 BSD-Lite release was made in July 1994 (FreeBSD 1.1.5.1).

FreeBSD was then reinvented based on 4.4BSD-Lite code, which was incomplete. FreeBSD 2.0 was released in November 1994. Later releases included 2.0.5 in June 1995, 2.1.5 in August 1996, 2.1.7.1 in February 1997, 2.2.1 in April 1997, 2.2.8 in November 1998, 3.0 in October 1998, 3.1 in February 1999, 3.2 in May 1999, 3.3 in September 1999, 3.4 in December 1999, 3.5 in June 2000, 4.0 in March 2000, 4.1 in July 2000, and 4.2 in November 2000.

The goal of the FreeBSD project is to provide software that can be used for any purpose with no strings attached. The idea is that the code will get the widest possible use and provide the most benefit. At present, it runs primarily on Intel platforms, although Alpha platforms are supported. Work is underway to port to other processor platforms as well.

## C.2 Design Principles

UNIX was designed to be a time-sharing system. The standard user interface (the shell) is simple and can be replaced by another, if desired. The file system is a multilevel tree, which allows users to create their own subdirectories. Each user data file is simply a sequence of bytes.

Disk files and I/O devices are treated as similarly as possible. Thus, device dependencies and peculiarities are kept in the kernel as much as possible. Even in the kernel, most of them are confined to the device drivers.

UNIX supports multiple processes. A process can easily create new processes. CPU scheduling is a simple priority algorithm. FreeBSD uses demand paging as a mechanism to support memory-management and CPU-scheduling decisions. Swapping is used if a system is suffering from excess paging.

Because UNIX was originated by Thompson and Ritchie as a system for their own convenience, it was small enough to understand. Most of the algorithms were selected for ***simplicity***, not for speed or sophistication. The intent was to have the kernel and libraries provide a small set of facilities that was sufficiently powerful to allow a person to build a more complex system if needed. UNIX's clean design has resulted in many imitations and modifications.

Although the designers of UNIX had a significant amount of knowledge about other operating systems, UNIX had no elaborate design spelled out before its implementation. This flexibility appears to have been one of the key factors in the development of the system. Some design principles were involved, however, even though they were not made explicit at the outset.

The UNIX system was designed by programmers for programmers. Thus, it has always been interactive, and facilities for program development have always been a high priority. Such facilities include the program ***make*** (which can be used to check which of a collection of source files for a program need to be compiled and then to do the compiling) and the ***Source Code Control System (SCCS)*** (which is used to keep successive versions of files available without having to store the entire contents of each step). The primary version-control system used by UNIX is the ***Concurrent Versions System*** (CVS) due to the large number of developers operating on and using the code.

The operating system is written mostly in C, which was developed to support UNIX, since neither Thompson nor Ritchie enjoyed programming in assembly language. The avoidance of assembly language was also necessary because of the uncertainty about the machines on which UNIX would be run. It has greatly simplified the problems of moving UNIX from one hardware system to another.

From the beginning, UNIX development systems have had all the UNIX sources available online, and the developers have used the systems under development as their primary systems. This pattern of development has greatly facilitated the discovery of deficiencies and their fixes, as well as of new possibilities and their implementations. It has also encouraged the plethora of UNIX variants existing today, but the benefits have outweighed the disadvantages. If something is broken, it can be fixed at a local site; there is no need to wait for the next release of the system. Such fixes, as well as new facilities, may be incorporated into later distributions.

The size constraints of the PDP-11 (and earlier computers used for UNIX) have forced a certain elegance. Where other systems have elaborate algorithms for dealing with pathological conditions, UNIX just does a controlled crash called ***panic***. Instead of attempting to cure such conditions, UNIX tries to prevent them. Where other systems would use brute force or macro-expansion, UNIX mostly has had to develop more subtle, or at least simpler, approaches.

These early strengths of UNIX produced much of its popularity, which in turn produced new demands that challenged those strengths. UNIX was used for tasks such as networking, graphics, and real-time operation, which did not always fit into its original text-oriented model. Thus, changes were made to certain internal facilities, and new programming interfaces were added. Supporting these new facilities and others—particularly window interfaces

—required large amounts of code, radically increasing the size of the system. For instance, both networking and windowing doubled the size of the system. This pattern in turn pointed out the continued strength of UNIX—whenever a new development occurred in the industry, UNIX could usually absorb it but remain UNIX.

## C.3 Programmer Interface

Like most operating systems, UNIX consists of two separable parts: the kernel and the systems programs. We can view the UNIX operating system as being layered, as shown in Figure C.2. Everything below the system-call interface and above the physical hardware is the ***kernel***. The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls. Systems programs use the kernel-supported system calls to provide useful functions, such as compilation and file manipulation.

System calls define the ***programmer interface*** to UNIX. The set of systems programs commonly available defines the ***user interface***. The programmer and user interface define the context that the kernel must support.

Most systems programs are written in C, and the ***UNIX Programmer's Manual*** presents all system calls as C functions. A system program written in C for FreeBSD on the Pentium can generally be moved to an Alpha FreeBSD system and simply recompiled, even though the two systems are quite different. The details of system calls are known only to the compiler. This feature is a major reason for the portability of UNIX programs.

System calls for UNIX can be roughly grouped into three categories: file manipulation, process control, and information manipulation. In Chapter 2, we listed a fourth category, device manipulation, but since devices in UNIX are treated as (special) files, the same system calls support both files and devices (although there is an extra system call for setting device parameters).

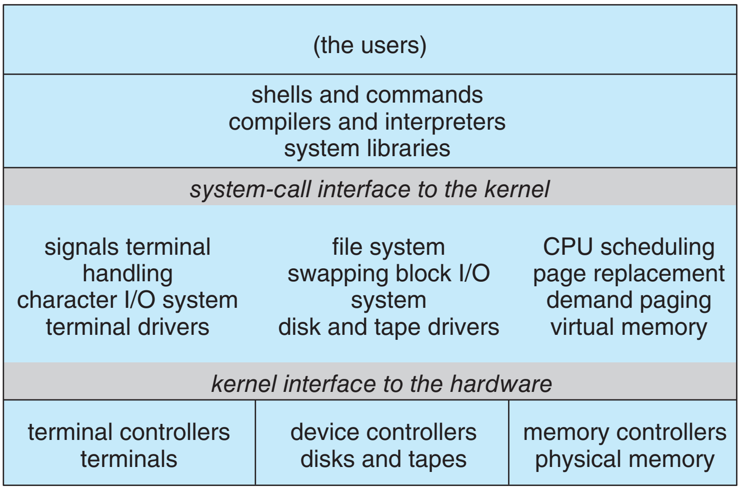

(the users)
shells and commands
compilers and interpreters
system libraries
system-call interface to the kernel
signals terminal handling
character I/O system
terminal drivers
file system
swapping block I/O system
disk and tape drivers
CPU scheduling
page replacement
demand paging
virtual memory
kernel interface to the hardware
terminal controllers
terminals
device controllers
disks and tapes
memory controllers
physical memory

**Figure C.2** 4.4BSD layer structure.

### C.3.1 File Manipulation

A ***file*** in UNIX is a sequence of bytes. Different programs expect various levels of structure, but the kernel does not impose a structure on files. For instance, the convention for text files is lines of ASCII characters separated by a single newline character (which is the linefeed character in ASCII), but the kernel knows nothing of this convention.

Files are organized in tree-structured ***directories***. Directories are themselves files that contain information on how to find other files. A ***path name*** to a file is a text string that identifies a file by specifying a path through the directory structure to the file. Syntactically, it consists of individual file-name elements separated by the slash character. For example, in */usr/local/font*, the first slash indicates the root of the directory tree, called the ***root directory***. The next element, *usr,* is a subdirectory of the root, *local* is a subdirectory of *usr*, and *font* is a file or directory in the directory ***local***. Whether *font* is an ordinary file or a directory cannot be determined from the path-name syntax.

The UNIX file system has both ***absolute path names*** and ***relative path names***. Absolute path names start at the root of the file system and are distinguished by a slash at the beginning of the path name; */usr/local/font* is an absolute path name. Relative path names start at the ***current directory***, which is an attribute of the process accessing the path name. Thus, *local/font* indicates a file or directory named *font* in the directory *local* in the current directory, which might or might not be */usr.*

A file may be known by more than one name in one or more directories. Such multiple names are known as ***links***, and all links are treated equally by the operating system. FreeBSD also supports ***symbolic links***, which are files containing the path name of another file. The two kinds of links are also known as ***hard links*** and ***soft links***. Soft (symbolic) links, unlike hard links, may point to directories and may cross file-system boundaries.

The file name "." in a directory is a hard link to the directory itself. The file name ".." is a hard link to the parent directory. Thus, if the current directory is */user/jlp/programs,* then *../bin/wdf* refers to */user/jlp/bin/wdf.*

Hardware devices have names in the file system. These ***device special files*** or ***special files*** are known to the kernel as device interfaces, but they are nonetheless accessed by the user by much the same system calls as are other files.

Figure C.3 shows a typical UNIX file system. The root (/) normally contains a small number of directories as well as */kernel,* the binary boot image of the operating system; */dev* contains the device special files, such as */dev/console, /dev/lp0, /dev/mt0,* and so on; and */bin* contains the binaries of the essential UNIX systems programs. Other binaries may be in */usr/bin* (for applications systems programs, such as text formatters), */usr/compat* (for programs from other operating systems, such as Linux), or */usr/local/bin* (for systems programs written at the local site). Library files—such as the C, Pascal, and FORTRAN subroutine libraries—are kept in */lib* (or */usr/lib* or */usr/local/lib*).

The files of users themselves are stored in a separate directory for each user, typically in */usr.* Thus, the user directory for *carol* would normally be in */usr/carol.* For a large system, these directories may be further grouped to ease administration, creating a file structure with */usr/prof/avi* and */usr/staff/carol.* Administrative files and programs, such as the password file, are kept in */etc.*

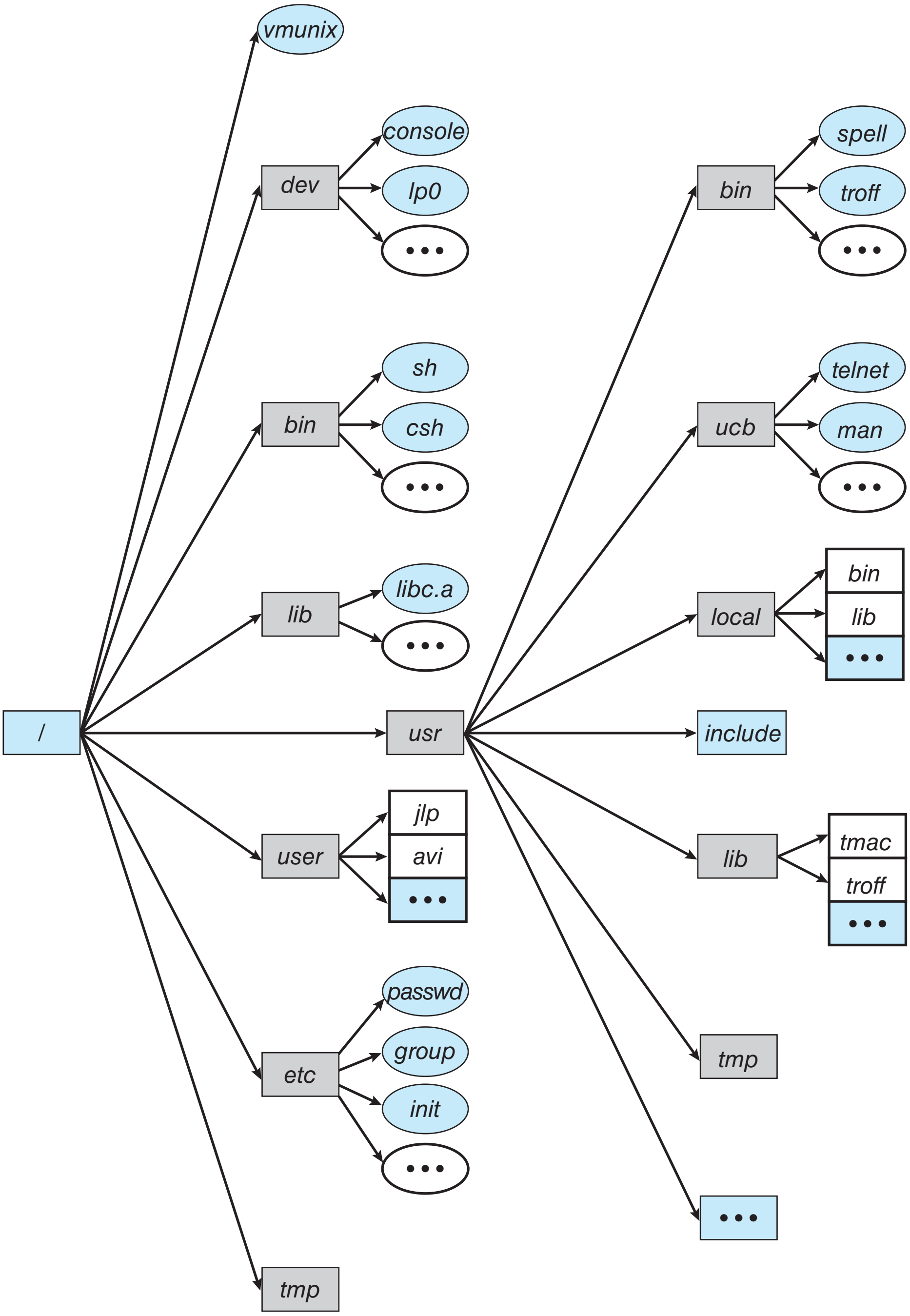

vmunix
dev
console
lp0
• • •
bin
sh
csh
• • •
lib
libc.a
• • •
/
usr
user
jlp
avi
• • •
etc
passwd
group
init
• • •
tmp
bin
spell
troff
• • •
ucb
telnet
man
• • •
local
bin
lib
• • •
include
lib
tmac
troff
• • •
tmp
• • •

**Figure C.3** Typical UNIX directory structure.

Temporary files can be put in */tmp*, which is normally erased during system boot, or in */usr/tmp*.

Each of these directories may have considerably more structure. For example, the font-description tables for the troff formatter for the Merganthaler 202 typesetter are kept in */usr/lib/troff/dev202*. All the conventions concerning the location of specific files and directories have been defined by programmers and their programs. The operating-system kernel needs only */etc/init*, which is used to initialize terminal processes, to be operable.

System calls for basic file manipulation are `creat()`, `open()`, `read()`, `write()`, `close()`, `unlink()`, and `trunc()`. The `creat()` system call, given a path name, creates an empty file (or truncates an existing one). An existing file is opened by the `open()` system call, which takes a path name and a mode (such as read, write, or read−write) and returns a small integer, called a ***file descriptor***. The file descriptor may then be passed to a `read()` or `write()` system call (along with a buffer address and the number of bytes to transfer) to perform data transfers to or from the file. A file is closed when its file descriptor is passed to the `close()` system call. The `trunc()` call reduces the length of a file to 0.

A file descriptor is an index into a small table of open files for this process. Descriptors start at 0 and seldom get higher than 6 or 7 for typical programs, depending on the maximum number of simultaneously open files.

Each `read()` or `write()` updates the current offset into the file, which is associated with the file-table entry and is used to determine the position in the file for the next `read()` or `write()`. The `lseek()` system call allows the position to be reset explicitly. It also allows the creation of sparse files (files with "holes" in them). The `dup()` and `dup2()` system calls can be used to produce a new file descriptor that is a copy of an existing one. The `fcntl()` system call can also do that and in addition can examine or set various parameters of an open file. For example, it can make each succeeding `write()` to an open file append to the end of that file. There is an additional system call, `ioctl()`, for manipulating device parameters. It can set the baud rate of a serial port or rewind a tape, for instance.

Information about the file (such as its size, protection modes, owner, and so on) can be obtained by the `stat()` system call. Several system calls allow some of this information to be changed: `rename()` (change file name), `chmod()` (change the protection mode), and `chown()` (change the owner and group). Many of these system calls have variants that apply to file descriptors instead of file names. The `link()` system call makes a hard link for an existing file, creating a new name for an existing file. A link is removed by the `unlink(())` system call; if it is the last link, the file is deleted. The `symlink()` system call makes a symbolic link.

Directories are made by the `mkdir()` system call and are deleted by `rmdir()`. The current directory is changed by `cd()`.

Although the standard file calls (`open()` and others) can be used on directories, it is inadvisable to do so, since directories have an internal structure that must be preserved. Instead, another set of system calls is provided to open a directory, to step through each file entry within the directory, to close the directory, and to perform other functions; these are `opendir()`, `readdir()`, `closedir()`, and others.

### C.3.2 Process Control

A ***process*** is a program in execution. Processes are identified by their ***process identifier***, which is an integer. A new process is created by the `fork()` system call. The new process consists of a copy of the address space of the original process (the same program and the same variables with the same values). Both processes (the parent and the child) continue execution at the instruction after the `fork()` with one difference: the return code for the `fork()` is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

Typically, the `execve()` system call is used after a fork by one of the two processes to replace that process's virtual memory space with a new program. The `execve()` system call loads a binary file into memory (destroying the memory image of the program containing the `execve()` system call) and starts its execution.

A process may terminate by using the `exit()` system call, and its parent process may wait for that event by using the `wait()` system call. If the child process crashes, the system simulates the `exit()` call. The `wait()` system call provides the process ID of a terminated child so that the parent can tell which of possibly many children terminated. A second system call, `wait3()`, is similar to `wait()` but also allows the parent to collect performance statistics about the child. Between the time the child exits and the time the parent completes one of the `wait()` system calls, the child is ***defunct***. A defunct process can do nothing but exists merely so that the parent can collect its status information. If the parent process of a defunct process exits before a child, the defunct process is inherited by the `init` process (which in turn waits on it) and becomes a ***zombie*** process. A typical use of these facilities is shown in Figure C.4.

The simplest form of communication between processes is by ***pipes***. A pipe may be created before the `fork()`, and its endpoints are then set up between the `fork()` and the `execve()`. A pipe is essentially a queue of bytes between two processes. The pipe is accessed by a file descriptor, like an ordinary file. One process writes into the pipe, and the other reads from the pipe. The size of the original pipe system was fixed by the system. With FreeBSD pipes are implemented on top of the socket system, which has variable-sized buffers. Reading from an empty pipe or writing into a full pipe causes the process to be blocked until the state of the pipe changes. Special arrangements are needed for a pipe to be placed between a parent and child (so only one is reading and one is writing).

All user processes are descendants of one original process, called `init` (which has process identifier 1). Each terminal port available for interactive use has a `getty` process forked for it by `init`. The `getty` process initializes terminal line parameters and waits for a user's login name, which it passes through

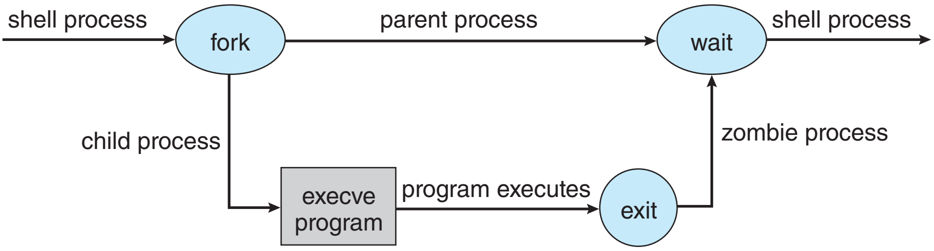


**Figure C.4** A shell forks a subprocess to execute a program.

an `execve()` as an argument to a `login` process. The `login` process collects the user's password, encrypts it, and compares the result to an encrypted string taken from the file */etc/passwd.* If the comparison is successful, the user is allowed to log in. The `login` process executes a ***shell***, or command interpreter, after setting the numeric ***user identifier*** of the process to that of the user logging in. (The shell and the user identifier are found in */etc/passwd* by the user's login name.) It is with this shell that the user ordinarily communicates for the rest of the login session. The shell itself forks subprocesses for the commands the user tells it to execute.

The user identifier is used by the kernel to determine the user's permissions for certain system calls, especially those involving file accesses. There is also a ***group identifier***, which is used to provide similar privileges to a collection of users. In FreeBSD a process may be in several groups simultaneously. The `login` process puts the shell in all the groups permitted to the user by the files */etc/passwd* and */etc/group.*

Two user identifiers are used by the kernel: the effective user identifier and the real user identifier. The ***effective user identifier*** is used to determine file access permissions. If the file of a program being loaded by an `execve()` has the `setuid` bit set in its inode, the effective user identifier of the process is set to the user identifier of the owner of the file, whereas the ***real user identifier*** is left as it was. This scheme allows certain processes to have more than ordinary privileges while still being executable by ordinary users. The setuid idea was patented by Dennis Ritchie (U.S. Patent 4,135,240) and is one of the distinctive features of UNIX. A similar `setgid` bit exists for groups. A process may determine its real and effective user identifier with the `getuid()` and `geteuid()` calls, respectively. The `getgid()` and `getegid()` calls determine the process's real and effective group identifier, respectively. The rest of a process's groups may be found with the `getgroups()` system call.

### C.3.3 Signals

***Signals*** are a facility for handling exceptional conditions similar to software interrupts. There are 20 different signals, each corresponding to a distinct condition. A signal may be generated by a keyboard interrupt, by an error in a process (such as a bad memory reference), or by a number of asynchronous events (such as timers or job-control signals from the shell). Almost any signal may also be generated by the `kill()` system call.

The `interrupt` signal, SIGINT, is used to stop a command before that command completes. It is usually produced by the ^C character (ASCII 3). As of 4.2BSD, the important keyboard characters are defined by a table for each terminal and can be redefined easily. The `quit` signal, SIGQUIT, is usually produced by the ^bs character (ASCII 28). The `quit` signal both stops the currently executing program and dumps its current memory image to a file named *core* in the current directory. The core file can be used by debuggers. SIGILL is produced by an illegal instruction and SIGSEGV by an attempt to address memory outside of the legal virtual memory space of a process.

Arrangements can be made either for most signals to be ignored (to have no effect) or for a routine in the user process (a signal handler) to be called. A signal handler may safely do one of two things before returning from catching a signal: call the `exit()` system call or modify a global variable. One signal (the `kill` signal, number 9, SIGKILL) cannot be ignored or caught by a signal handler. SIGKILL is used, for example, to kill a runaway process that is ignoring other signals such as SIGINT and SIGQUIT.

Signals can be lost. If another signal of the same kind is sent before a previous signal has been accepted by the process to which it is directed, the first signal will be overwritten, and only the last signal will be seen by the process. In other words, a call to the signal handler tells a process that there has been at least one occurrence of the signal. Also, there is no relative priority among UNIX signals. If two different signals are sent to the same process at the same time, we cannot know which one the process will receive first.

Signals were originally intended to deal with exceptional events. As is true of most UNIX features, however, signal use has steadily expanded. 4.1BSD introduced job control, which uses signals to start and stop subprocesses on demand. This facility allows one shell to control multiple processes—starting, stopping, and backgrounding them as the user wishes. The SIGWINCH signal, invented by Sun Microsystems, is used for informing a process that the window in which output is being displayed has changed size. Signals are also used to deliver urgent data from network connections.

Users wanted more reliable signals and a bug fix in an inherent race condition in the old signal implementation. Thus, 4.2BSD brought with it a race-free, reliable, separately implemented signal capability. It allows individual signals to be blocked during critical sections, and it has a new system call to let a process sleep until interrupted. It is similar to hardware-interrupt functionality. This capability is now part of the POSIX standard.

### C.3.4 Process Groups

Groups of related processes frequently cooperate to accomplish a common task. For instance, processes may create, and communicate over, pipes. Such a set of processes is termed a ***process group***, or a ***job***. Signals may be sent to all processes in a group. A process usually inherits its process group from its parent, but the `setpgrp()` system call allows a process to change its group.

Process groups are used by the C shell to control the operation of multiple jobs. Only one process group may use a terminal device for I/O at any time. This ***foreground*** job has the attention of the user on that terminal, while all other nonattached jobs (***background*** jobs) perform their functions without user interaction. Access to the terminal is controlled by process group signals. Each job has a ***controlling terminal*** (again, inherited from its parent). If the process group of the controlling terminal matches the group of a process, that process is in the foreground and is allowed to perform I/O. If a nonmatching (background) process attempts the same, a SIGTTIN or SIGTTOU signal is sent to its process group. This signal usually causes the process group to freeze until it is foregrounded by the user, at which point it receives a SIGCONT signal, indicating that the process can perform the I/O. Similarly, a SIGSTOP may be sent to the foreground process group to freeze it.

### C.3.5 Information Manipulation

System calls exist to set and return both an interval timer (`getitimer()`/`setitimer()`) and the current time (`gettimeofday()`/`settimeofday()`) in microseconds. In addition, processes can ask for their process identifier (`getpid()`), their group identifier (`getgid()`), the name of the machine on which they are executing (`gethostname()`), and many other values.

### C.3.6 Library Routines

The system-call interface to UNIX is supported and augmented by a large collection of library routines and header files. The header files provide the definition of complex data structures used in system calls. In addition, a large library of functions provides additional program support.

For example, the UNIX I/O system calls provide for the reading and writing of blocks of bytes. Some applications may want to read and write only 1 byte at a time. Although possible, that would require a system call for each byte—a very high overhead. Instead, a set of standard library routines (the standard I/O package accessed through the header file <*stdio.h*>) provides another interface, which reads and writes several thousand bytes at a time using local buffers and transfers between these buffers (in user memory) when I/O is desired. Formatted I/O is also supported by the standard I/O package.

Additional library support is provided for mathematical functions, network access, data conversion, and so on. The FreeBSD kernel supports over 300 system calls; the C program library has over 300 library functions. The library functions eventually result in system calls where necessary (for example, the `getchar()` library routine will result in a `read()` system call if the file buffer is empty). However, the programmer generally does not need to distinguish between the basic set of kernel system calls and the additional functions provided by library functions.

## C.4 User Interface

Both the programmer and the user of a UNIX system deal mainly with the set of systems programs that have been written and are available for execution. These programs make the necessary system calls to support their function, but the system calls themselves are contained within the program and do not need to be obvious to the user.

The common systems programs can be grouped into several categories; most of them are file or directory oriented. For example, the systems programs to manipulate directories are `mkdir` to create a new directory, `rmdir` to remove a directory, `cd` to change the current directory to another, and `pwd` to print the absolute path name of the current (working) directory.

The `ls` program lists the names of the files in the current directory. Any of 28 options can ask that properties of the files be displayed also. For example, the `-l` option asks for a long listing showing the file name, owner, protection, date and time of creation, and size. The `cp` program creates a new file that is a copy of an existing file. The `mv` program moves a file from one place to another in the directory tree. In most cases, this move simply requires a renaming of the file. If necessary, however, the file is copied to the new location, and the old copy is deleted. A file is deleted by the `rm` program (which makes an `unlink()` system call).

To display a file on the terminal, a user can run `cat`. The `cat` program takes a list of files and concatenates them, copying the result to the standard output, commonly the terminal. On a high-speed cathode-ray tube (CRT) display, of course, the file may speed by too fast to be read. The `more` program displays the file one screen at a time, pausing until the user types a character to continue to the next screen. The `head` program displays just the first few lines of a file; `tail` shows the last few lines.

These are the basic systems programs widely used in UNIX. In addition, there are a number of editors (`ed`, `sed`, `emacs`, `vi`, and so on), compilers (C, python, FORTRAN, and so on), and text formatters (troff, TEX, scribe, and so on). There are also programs for sorting (`sort`) and comparing files (`cmp`, `diff`), looking for patterns (`grep`, `awk`), sending mail to other users (`mail`), and many other activities.

### C.4.1 Shells and Commands

Both user-written and systems programs are normally executed by a command interpreter. The command interpreter in UNIX is a user process like any other. As noted earlier, it is called a shell—because it surrounds the kernel of the operating system. Users can write their own shells, and, in fact, several shells are in general use. The ***Bourne shell***, written by Steve Bourne, is probably the most widely used—or, at least, the most widely available. The ***C shell***, mostly the work of Bill Joy, a founder of Sun Microsystems, is the most popular on BSD systems. The Korn shell, by Dave Korn, has become popular because it combines the features of the Bourne shell and the C shell.

The common shells share much of their command-language syntax. UNIX is normally an interactive system. The shell indicates its readiness to accept another command by typing a prompt, and the user types a command on a single line. For instance, in the line

```
% ls -l
```

the percent sign is the usual C shell prompt, and the `ls -l` (typed by the user) is the (long) list-directory command. Commands can take arguments, which the user types after the command name on the same line, separated by white space (spaces or tabs).

Although a few commands are built into the shells (such as `cd`), a typical command is an executable binary object file. A list of several directories, the ***search path***, is kept by the shell. For each command, each of the directories in the search path is searched, in order, for a file of the same name. If a file is found, it is loaded and executed. The search path can be set by the user. The directories */bin* and */usr/bin* are almost always in the search path, and a typical search path on a FreeBSD system might be

*( . /usr/avi/bin /usr/local/bin /bin /usr/bin )*

The `ls` command's object file is */bin/ls*, and the shell itself is */bin/sh* (the Bourne shell) or */bin/csh* (the C shell).

Execution of a command is done by a `fork()` system call followed by an `execve()` of the object file. The shell usually then does a `wait()` to suspend its own execution until the command completes (Figure C.4). There is a simple syntax (an ampersand [&] at the end of the command line) to indicate that the shell should *not* wait for the completion of the command. A command left running in this manner while the shell continues to interpret further commands is said to be a ***background*** command, or to be running in the background. Processes for which the shell *does* wait are said to run in the ***foreground***.

The C shell in FreeBSD systems provides a facility called ***job control*** (partially implemented in the kernel), as mentioned previously. Job control allows processes to be moved between the foreground and the background. The processes can be stopped and restarted on various conditions, such as a background job wanting input from the user's terminal. This scheme allows most of the control of processes provided by windowing or layering interfaces but requires no special hardware. Job control is also useful in window systems, such as the X Window System developed at MIT. Each window is treated as a terminal, allowing multiple processes to be in the foreground (one per window) at any one time. Of course, background processes may exist on any of the windows. The Korn shell also supports job control, and job control (and process groups) will likely be standard in future versions of UNIX.

### C.4.2 Standard I/O

Processes can open files as they like, but most processes expect three file descriptors (numbers 0, 1, and 2) to be open when they start. These file descriptors are inherited across the `fork()` (and possibly the `execve()`) that created the process. They are known as ***standard input*** (0), ***standard output*** (1), and ***standard error*** (2). All three are frequently open to the user's terminal. Thus, the program can read what the user types by reading standard input, and the program can send output to the user's screen by writing to standard output. The standard-error file descriptor is also open for writing and is used for error output; standard output is used for ordinary output. Most programs can also accept a file (rather than a terminal) for standard input and standard output. The program does not care where its input is coming from and where its output is going. This is one of the elegant design features of UNIX.

The common shells have a simple syntax for changing what files are open for the standard I/O streams of a process. Changing a standard file is called ***I/O redirection***. The syntax for I/O redirection is shown in Figure C.5. In this

| command | meaning of command |
|---|---|
| % *ls* > *filea* | direct output of *ls* to file *filea* |
| % *pr* < *filea* > *fileb* | input from *filea* and output to *fileb* |
| % *lpr* < *fileb* | input from *fileb* |
| % % make program > & errs | save both standard output and standard error in a file |

**Figure C.5** Standard /io/ redirection.

example, the `ls` command produces a listing of the names of files in the current directory, the `pr` command formats that list into pages suitable for a printer, and the `lpr` command spools the formatted output to a printer, such as */dev/lp0*. The subsequent command forces all output and all error messages to be redirected to a file. Without the ampersand, error messages appear on the terminal.

### C.4.3 Pipelines, Filters, and Shell Scripts

The first three commands of Figure C.5 could have been coalesced into the one command

% ***ls | pr | lpr***

Each vertical bar tells the shell to arrange for the output of the preceding command to be passed as input to the following command. A **pipe** is used to carry the data from one process to the other. One process writes into one end of the pipe, and another process reads from the other end. In the example, the write end of one pipe would be set up by the shell to be the standard output of `ls`, and the read end of the pipe would be the standard input of `pr`. Another pipe would be between `pr` and `lpr`.

A command such as `pr` that passes its standard input to its standard output, performing some processing on it, is called a ***filter***. Many UNIX commands can be used as filters. Complicated functions can be pieced together as pipelines of common commands. Also, common functions, such as output formatting, do not need to be built into numerous commands, because the output of almost any program can be piped through `pr` (or some other appropriate filter).

Both of the common UNIX shells are also programming languages, with shell variables and the usual higher-level programming-language control constructs (loops, conditionals). The execution of a command is analogous to a subroutine call. A file of shell commands, a ***shell script***, can be executed like any other command, with the appropriate shell being invoked automatically to read it. ***Shell programming*** thus can be used to combine ordinary programs conveniently for sophisticated applications without the need for any programming in conventional languages.

This external user view is commonly thought of as the definition of UNIX, yet it is the most easily changed definition. Writing a new shell with a quite different syntax and semantics would greatly change the user view while not changing the kernel or even the programmer interface. Several menu-driven and iconic interfaces for UNIX exist, and the X Window System is rapidly becoming a standard. The heart of UNIX is, of course, the kernel. This kernel is much more difficult to change than is the user interface, because all programs depend on the system calls that it provides to remain consistent. Of course, new system calls can be added to increase functionality, but programs must then be modified to use the new calls.

## C.5 Process Management

A major design problem for operating systems is the representation of processes. One substantial difference between UNIX and many other systems is the ease with which multiple processes can be created and manipulated. These processes are represented in UNIX by various control blocks. No system control blocks are accessible in the virtual address space of a user process; control blocks associated with a process are stored in the kernel. The kernel uses the information in these control blocks for process control and CPU scheduling.

### C.5.1 Process Control Blocks

The most basic data structure associated with processes is the ***process structure***. A process structure contains everything that the system needs to know about a process when the process is swapped out, such as its unique process identifier, scheduling information (for example, the priority of the process), and pointers to other control blocks. There is an array of process structures whose length is defined at system-linking time. The process structures of ready processes are kept linked together by the scheduler in a doubly linked list (the ready queue), and there are pointers from each process structure to the process's parent, to its youngest living child, and to various other relatives of interest, such as a list of processes sharing the same program code (text).

The ***virtual address space*** of a user process is divided into text (program code), data, and stack segments. The data and stack segments are always in the same address space, but they may grow separately, and usually in opposite directions. Most frequently, the stack grows down as the data grow up toward it. The text segment is sometimes (as on an Intel 8086 with separate instruction and data space) in an address space different from the data and stack, and it is usually read-only. The debugger puts a text segment in read–write mode to allow insertion of breakpoints.

Every process with sharable text (almost all, under FreeBSD) has a pointer from its process structure to a ***text structure***. The text structure records how many processes are using the text segment, including a pointer into a list of their process structures, and where the page table for the text segment can be found on disk when it is swapped. The text structure itself is always resident in main memory. An array of such structures is allocated at system link time. The text, data, and stack segments for the processes may be swapped. When the segments are swapped in, they are paged.

The ***page tables*** record information on the mapping from the process's virtual memory to physical memory. The process structure contains pointers to the page table, for use when the process is resident in main memory, or the address of the process on the swap device, when the process is swapped. There is no special separate page table for a shared text segment; every process sharing the text segment has entries for its pages in the process's page table.

Information about the process needed only when the process is resident (that is, not swapped out) is kept in the ***user structure*** (or ***u structure***), rather than in the process structure. This structure is mapped read-only into user virtual address space, so user processes can read its contents. It is writable by the kernel. The user structure contains a copy of the process control block, or PCB, which is kept here for saving the process's general registers, stack pointer, program counter, and page-table base registers when the process is not running. There is space to keep system-call parameters and return values. All user and group identifiers associated with the process (not just the effective user identifier kept in the process structure) are kept here. Signals, timers, and quotas have data structures here. Of more obvious relevance to the ordinary user, the current directory and the table of open files are maintained in the user structure.

Every process has both a user and a system mode. Most ordinary work is done in ***user mode***, but when a system call is made, it is performed in ***system mode***. The system and user phases of a process never execute simultaneously. When a process is executing in system mode, a ***kernel stack*** for that process is used, rather than the user stack belonging to that process. The kernel stack for the process immediately follows the user structure. The kernel stack and the user structure together compose the ***system data segment*** for the process. The kernel has its own stack for use when it is not doing work on behalf of a process (for instance, for interrupt handling).

Figure C.6 illustrates how the process structure is used to find the various parts of a process.

The `fork()` system call allocates a new process structure (with a new process identifier) for the child process and copies the user structure. There is ordinarily no need for a new text structure, as the processes share their text. The appropriate counters and lists are merely updated. A new page table is constructed, and new main memory is allocated for the data and stack segments of the child process. The copying of the user structure preserves open file descriptors, user and group identifiers, signal handling, and most similar properties of a process.

The `vfork()` system call does *not* copy the data and stack to the new process; rather, the new process simply shares the page table of the old one. A new user structure and a new process structure are still created. A common use of this system call occurs when a shell executes a command and waits for its completion. The parent process uses `vfork()` to produce the child process. Because the child process wishes to use an `execve()` immediately to change its virtual address space completely, there is no need for a complete copy of the

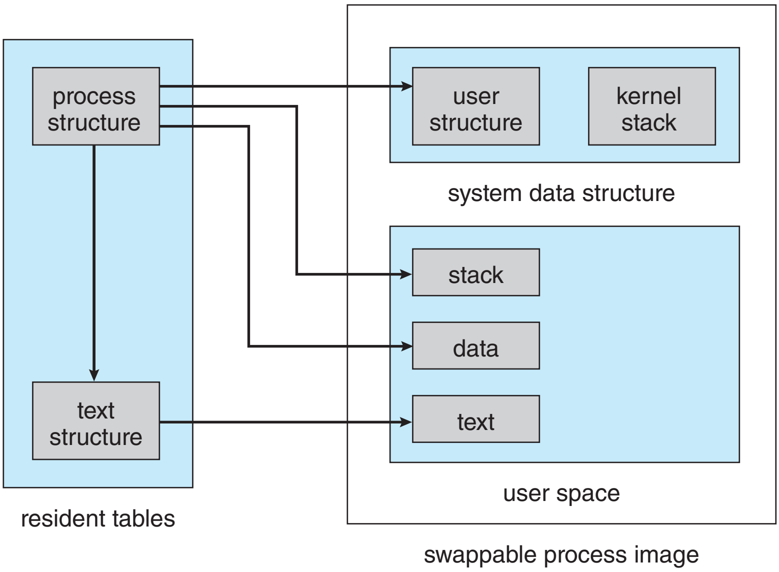


**Figure C.6** Finding parts of a process using the process structure.

parent process. Such data structures as are necessary for manipulating pipes may be kept in registers between the `vfork()` and the `execve()`. Files may be closed in one process without affecting the other process, since the kernel data structures involved depend on the user structure, which is not shared. The parent is suspended when it calls `vfork()` until the child either calls `execve()` or terminates, so that the parent will not change memory that the child needs.

When the parent process is large, `vfork()` can produce substantial savings in system CPU time. However, it is a fairly dangerous system call, since any memory change occurs in both processes until the `execve()` occurs. An alternative is to share all pages by duplicating the page table but to mark the entries of both page tables as ***copy-on-write***. The hardware protection bits are set to trap any attempt to write in these shared pages. If such a trap occurs, a new frame is allocated, and the shared page is copied to the new frame. The page tables are adjusted to show that this page is no longer shared (and therefore no longer needs to be write-protected), and execution can resume.

An `execve()` system call creates no new process or user structure. Rather, the text and data of the process are replaced. Open files are preserved (although there is a way to specify that certain file descriptors are to be closed on an `execve()`). Most signal-handling properties are preserved, but arrangements to call a specific user routine on a signal are canceled, for obvious reasons. The process identifier and most other properties of the process are unchanged.

### C.5.2 CPU Scheduling

***CPU scheduling*** in UNIX is designed to benefit interactive processes. Processes are given small CPU time slices by a priority algorithm that reduces to round-robin scheduling for CPU-bound jobs.

Every process has a ***scheduling priority*** associated with it; larger numbers indicate lower priority. Processes doing disk I/O or other important tasks have priorities less than “pzero” and cannot be killed by signals. Ordinary user processes have positive priorities and thus are less likely to be run than is any system process, although user processes can set precedence over one another through the `nice` command.

The more CPU time a process accumulates, the lower (more positive) its priority becomes, and vice versa. This negative feedback in CPU scheduling makes it difficult for a single process to take all the CPU time. Process aging is employed to prevent starvation.

Older UNIX systems used a 1-second quantum for the round-robin scheduling. FreeBSD reschedules processes every 0.1 second and recomputes priorities every second. The round-robin scheduling is accomplished by the ***timeout*** mechanism, which tells the clock interrupt driver to call a kernel subroutine after a specified interval. The subroutine to be called in this case causes the rescheduling and then resubmits a timeout to call itself again. The priority recomputation is also timed by a subroutine that resubmits a timeout for itself.

There is no preemption of one process by another in the kernel. A process may relinquish the CPU because it is waiting for I/O or because its time slice has expired. When a process chooses to relinquish the CPU, it goes to ***sleep*** on an ***event***. The kernel primitive used for this purpose is called `sleep()` (not to be confused with the user-level library routine of the same name). `Sleep()` takes an argument that is, by convention, the address of a kernel data structure related to an event for which a process is waiting. When the event occurs, the system process that knows about it calls `wakeup()` with the address corresponding to the event, and *all* processes that had done a sleep on the same address are put in the ready queue to be run.

For example, a process waiting for disk I/O to complete will sleep on the address of the buffer header corresponding to the data being transferred. When the interrupt routine for the disk driver notes that the transfer is complete, it calls `wakeup()` on the buffer header. The interrupt uses the kernel stack for whatever process happened to be running at the time, and the `wakeup()` is done from that system process.

The process that actually does run is chosen by the scheduler. `Sleep()` takes a second argument, which is the scheduling priority to be used for this purpose. This priority argument, if less than "pzero," also prevents the process from being awakened prematurely by some exceptional event, such as a signal.

When a signal is generated, it is left pending until the system half of the affected process next runs. This event usually happens soon, since the signal normally causes the process to be awakened if the process has been waiting for some other condition.

No memory is associated with events. The caller of the routine that does a `sleep()` on an event must be prepared to deal with a premature return, including the possibility that the reason for waiting has vanished.

***Race conditions*** are involved in the event mechanism. If a process decides (because of checking a flag in memory, for instance) to sleep on an event, and the event occurs before the process can execute the primitive that does the `sleep()` on the event, the process sleeping may then sleep forever. We prevent this situation by raising the hardware processor priority during the critical section so that no interrupts can occur. Thus, only the process desiring the event can run until it is sleeping. Hardware processor priority is used in this manner to protect critical regions throughout the kernel and is the greatest obstacle to porting UNIX to multiple-processor machines. However, this problem has not prevented such porting from being done repeatedly.

Many processes, such as text editors, are I/O bound and are, in general, scheduled mainly on the basis of waiting for I/O. Experience suggests that the UNIX scheduler performs best with I/O-bound jobs, as can be observed when several CPU-bound jobs, such as text formatters or language interpreters, are running.

What has been referred to here as ***CPU scheduling*** corresponds closely to the ***short-term scheduling*** of Chapter 3. However, the negative-feedback property of the priority scheme provides some long-term scheduling in that it largely determines the long-term ***job mix***. Medium-term scheduling is done by the swapping mechanism described in Section C.6.

## C.6 Memory Management

Much of UNIX's early development was done on a PDP-11. The PDP-11 has only eight segments in its virtual address space, and the size of each is at most 8,192 bytes. The larger machines, such as the PDP-11/70, allow separate instruction and address spaces, effectively doubling the address space and number of segments, but this address space is still relatively small. In addition, the kernel was even more severely constrained due to dedication of one data segment to interrupt vectors, another to point at the per-process system data segment, and yet another for the UNIBUS (system I/O bus) registers. Further, on the smaller PDP-11s, total physical memory was limited to 256 KB. The total memory resources were insufficient to justify or support complex memory-management algorithms. Thus, UNIX swapped entire process memory images.

Berkeley introduced paging to UNIX with 3BSD. VAX 4.2BSD is a demand-paged virtual memory system. Paging eliminates external fragmentation of memory. (Internal fragmentation still occurs, but it is negligible with a reasonably small page size.) Because paging allows execution with only parts of each process in memory, more jobs can be kept in main memory, and swapping can be kept to a minimum. ***Demand paging*** is done in a straightforward manner. When a process needs a page and the page is not there, a page fault to the kernel occurs, a frame of main memory is allocated, and the proper disk page is read into the frame.

There are a few optimizations. If the page needed is still in the page table for the process but has been marked invalid by the page-replacement process, it can be marked valid and used without any I/O transfer. Pages can similarly be retrieved from the list of free frames. When most processes are started, many of their pages are prepaged and are put on the free list for recovery by this mechanism. Arrangements can also be made for a process to have no prepaging on startup. That is seldom done, however, because it results in more page-fault overhead, being closer to pure demand paging. FreeBSD implements page coloring with paging queues. The queues are arranged according to the size of the processor's L1 and L2 caches. When a new page needs to be allocated, FreeBSD tries to get one that is optimally aligned for the cache. If the page has to be fetched from disk, it must be locked in memory for the duration of the transfer. This locking ensures that the page will not be selected for page replacement. Once the page is fetched and mapped properly, it must remain locked if raw physical I/O is being done on it.

The page-replacement algorithm is more interesting. 4.2BSD uses a modification of the ***second-chance (clock) algorithm*** described in Section 10.4.5. The map of all nonkernel main memory (the ***core map*** or ***cmap***) is swept linearly and repeatedly by a software ***clock hand***. When the clock hand reaches a given frame, if the frame is marked as being in use by some software condition (for example, if physical I/O is in progress using it) or if the frame is already free, the frame is left untouched, and the clock hand sweeps to the next frame. Otherwise, the corresponding text or process page-table entry for this frame is located. If the entry is already invalid, the frame is added to the free list. Otherwise, the page-table entry is made invalid but reclaimable (that is, if it has not been paged out by the next time it is wanted, it can just be made valid again).

BSD Tahoe added support for systems that implement the reference bit. On such systems, one pass of the clock hand turns the reference bit off, and a second pass places those pages whose reference bits remain off onto the free list for replacement. Of course, if the page is dirty, it must first be written to disk before being added to the free list. Pageouts are done in clusters to improve performance.

There are checks to make sure that the number of valid data pages for a process does not fall too low and to keep the paging device from being flooded with requests. There is also a mechanism by which a process can limit the amount of main memory it uses.

The LRU clock-hand scheme is implemented in the `pagedaemon`, which is process 2. (Remember that the `swapper` is process 0 and `init` is process 1.) This process spends most of its time sleeping, but a check is done several times per second (scheduled by a timeout) to see if action is necessary. If it is, process 2 is awakened. Whenever the number of free frames falls below a threshold, `lotsfree`, the `pagedaemon` is awakened. Thus, if there is always a large amount of free memory, the `pagedaemon` imposes no load on the system, because it never runs.

The sweep of the clock hand each time the `pagedaemon` process is awakened (that is, the number of frames scanned, which is usually more than the number paged out) is determined both by the number of frames lacking to reach `lotsfree` and by the number of frames that the scheduler has determined are needed for various reasons (the more frames needed, the longer the sweep). If the number of frames free rises to `lotsfree` before the expected sweep is completed, the hand stops, and the `pagedaemon` process sleeps. The parameters that control the range of the clock-hand sweep are determined at system startup according to the amount of main memory, such that `pagedaemon` does not use more than 10 percent of all CPU time.

If the scheduler decides that the paging system is overloaded, processes will be swapped out whole until the overload is relieved. This swapping usually happens only if several conditions are met: load average is high; free memory has fallen below a low limit, `minfree`; and the average memory available over recent time is less than a desirable amount, `desfree`, where `lotsfree > desfree > minfree`. In other words, only a chronic shortage of memory with several processes trying to run will cause swapping, and even then free memory has to be extremely low at the moment. (An excessive paging rate or a need for memory by the kernel itself may also enter into the calculations, in rare cases.) Processes may be swapped by the scheduler, of course, for other reasons (such as simply because they have not run for a long time).

The parameter `lotsfree` is usually one-quarter of the memory in the map that the clock hand sweeps, and `desfree` and `minfree` are usually the same across different systems but are limited to fractions of available memory. FreeBSD dynamically adjusts its paging queues so these virtual memory parameters will rarely need to be adjusted. `Minfree` pages must be kept free in order to supply any pages that might be needed at interrupt time.

Every process's text segment is, by default, shared and read-only. This scheme is practical with paging, because there is no external fragmentation, and the swap space gained by sharing more than offsets the negligible amount of overhead involved, as the kernel virtual space is large.

CPU scheduling, memory swapping, and paging interact. The lower the priority of a process, the more likely that its pages will be paged out and the more likely that it will be swapped in its entirety. The age preferences in choosing processes to swap guard against thrashing, but paging does so more effectively. Ideally, processes will not be swapped out unless they are idle, because each process will need only a small working set of pages in main memory at any one time, and the `pagedaemon` will reclaim unused pages for use by other processes.

The amount of memory the process will need is some fraction of that process's total virtual size—up to one-half if that process has been swapped out for a long time.

## C.7 File System

The UNIX file system supports two main objects: files and directories. Directories are just files with a special format, so the representation of a file is the basic UNIX concept.

### C.7.1 Blocks and Fragments

Most of the file system is taken up by ***data blocks***, which contain whatever the users have put in their files. Let's consider how these data blocks are stored on the disk.

The hardware disk sector is usually 512 bytes. A block size larger than 512 bytes is desirable for speed. However, because UNIX file systems usually contain a very large number of small files, much larger blocks would cause excessive internal fragmentation. That is why the earlier 4.1BSD file system was limited to a 1,024-byte (1-KB) block. The 4.2BSD solution is to use *two* block sizes for files that have no indirect blocks. All the blocks of a file are of a large size (such as 8 KB) except the last. The last block is an appropriate multiple of a smaller fragment size (for example, 1,024 KB) to fill out the file. Thus, a file of size 18,000 bytes would have two 8-KB blocks and one 2-KB fragment (which would not be filled completely).

The block and fragment sizes are set during file-system creation according to the intended use of the file system. If many small files are expected, the fragment size should be small; if repeated transfers of large files are expected, the basic block size should be large. Implementation details force a maximum block-to-fragment ratio of 8:1 and a minimum block size of 4 KB, so typical choices are 4,096:512 for the former case and 8,192:1,024 for the latter.

Suppose data are written to a file in transfer sizes of 1-KB bytes, and the block and fragment sizes of the file system are 4 KB and 512 bytes. The file system will allocate a 1-KB fragment to contain the data from the first transfer. The next transfer will cause a new 2-KB fragment to be allocated. The data from the original fragment must be copied into this new fragment, followed by the second 1-KB transfer. The allocation routines attempt to find the required space on the disk immediately following the existing fragment so that no copying is necessary. If they cannot do so, up to seven copies may be required before the fragment becomes a block. Provisions have been made for programs to discover the block size for a file so that transfers of that size can be made, to avoid fragment recopying.

### C.7.2 Inodes

A file is represented by an ***inode***, which is a record that stores most of the information about a specific file on the disk. (See Figure C.7.) The name *inode* (pronounced *EYE node*) is derived from "index node" and was originally spelled "i-node"; the hyphen fell out of use over the years. The term is sometimes spelled "I node."

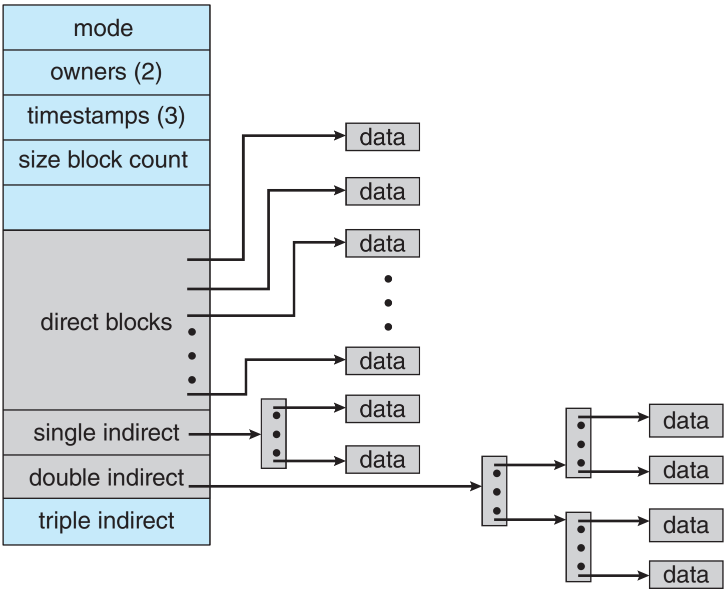


**Figure C.7** The UNIX inode.

The inode contains the user and group identifiers of the file, the times of the last file modification and access, a count of the number of hard links (directory entries) to the file, and the type of the file (plain file, directory, symbolic link, character device, block device, or socket). In addition, the inode contains 15 pointers to the disk blocks containing the data contents of the file. The first 12 of these pointers point to ***direct blocks***. That is, they contain addresses of blocks that contain data of the file. Thus, the data for small files (no more than 12 blocks) can be referenced immediately, because a copy of the inode is kept in main memory while a file is open. If the block size is 4 KB, then up to 48 KB of data can be accessed directly from the inode.

The next three pointers in the inode point to ***indirect blocks***. If the file is large enough to use indirect blocks, each of the indirect blocks is of the major block size; the fragment size applies only to data blocks. The first indirect block pointer is the address of a ***single indirect block***. The single indirect block is an index block containing not data but the addresses of blocks that do contain data. Then, there is a ***double-indirect-block pointer***, the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks. The last pointer would contain the address of a ***triple indirect block***; however, there is no need for it.

The minimum block size for a file system in 4.2BSD is 4 KB, so files with as many as $2^{32}$ bytes will use only double, not triple, indirection. That is, since each block pointer takes 4 bytes, we have 49,152 bytes accessible in direct blocks, 4,194,304 bytes accessible by a single indirection, and 4,294,967,296 bytes reachable through double indirection, for a total of 4,299,210,752 bytes, which is larger than $2^{32}$ bytes. The number $2^{32}$ is significant because the file offset in the file structure in main memory is kept in a 32-bit word. Files therefore cannot be larger than $2^{32}$ bytes. Since file pointers are signed integers (for seeking backward and forward in a file), the actual maximum file size is $2^{32-1}$ bytes. Two gigabytes is large enough for most purposes.

### C.7.3 Directories

Plain files are not distinguished from directories at this level of implementation. Directory contents are kept in data blocks, and directories are represented by an inode in the same way as plain files. Only the inode type field distinguishes between plain files and directories. Plain files are not assumed to have a structure, whereas directories have a specific structure. In Version 7, file names were limited to 14 characters, so directories were a list of 16-byte entries: 2 bytes for an inode number and 14 bytes for a file name.

In FreeBSD file names are of variable length, up to 255 bytes, so directory entries are also of variable length. Each entry contains first the length of the entry, then the file name and the inode number. This variable-length entry makes the directory management and search routines more complex, but it allows users to choose much more meaningful names for their files and directories. The first two names in every directory are "." and "..". New directory entries are added to the directory in the first space available, generally after the existing files. A linear search is used.

The user refers to a file by a path name, whereas the file system uses the inode as its definition of a file. Thus, the kernel has to map the supplied user path name to an inode. The directories are used for this mapping.

First, a starting directory is determined. As mentioned earlier, if the first character of the path name is "/," the starting directory is the root directory. If the path name starts with any character other than a slash, the starting directory is the current directory of the current process. The starting directory is checked for proper file type and access permissions, and an error is returned if necessary. The inode of the starting directory is always available.

The next element of the path name, up to the next "/" or to the end of the path name, is a file name. The starting directory is searched for this name, and an error is returned if the name is not found. If the path name has yet another element, the current inode must refer to a directory; an error is returned if it does not or if access is denied. This directory is searched in the same way as the previous one. This process continues until the end of the path name is reached and the desired inode is returned. This step-by-step process is needed because at any directory a mount point (or symbolic link, as discussed below) may be encountered, causing the translation to move to a different directory structure for continuation.

Hard links are simply directory entries like any other. We handle symbolic links for the most part by starting the search over with the path name taken from the contents of the symbolic link. We prevent infinite loops by counting the number of symbolic links encountered during a path-name search and returning an error when a limit (eight) is exceeded.

Nondisk files (such as devices) do not have data blocks allocated on the disk. The kernel notices these file types (as indicated in the inode) and calls appropriate drivers to handle I/O for them.

Once the inode is found by, for instance, the `open()` system call, a ***file structure*** is allocated to point to the inode. The file descriptor given to the user refers to this file structure. FreeBSD has a ***directory name cache*** to hold

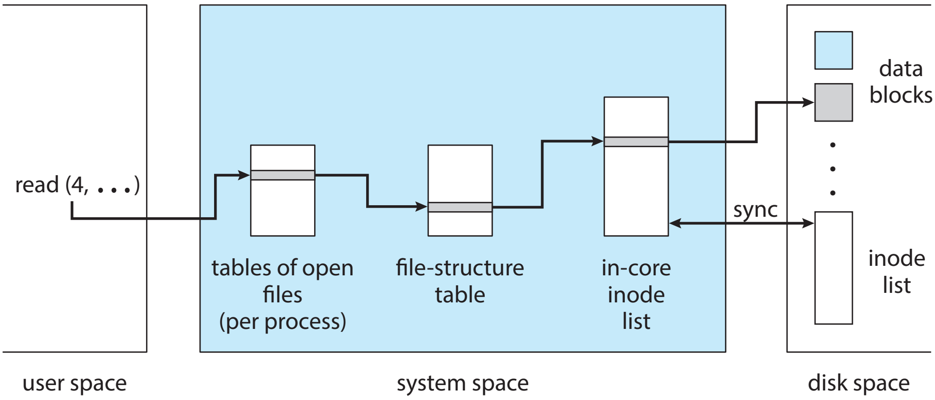


**Figure C.8** File-system control blocks.

recent directory-to-inode translations, which greatly increases file-system performance.

### C.7.4 Mapping a File Descriptor to an Inode

A system call that refers to an open file indicates the file by passing a file descriptor as an argument. The file descriptor is used by the kernel to index a table of open files for the current process. Each entry in the table contains a pointer to a file structure. This file structure in turn points to the inode; see Figure C.8. The open file table has a fixed length, which is settable only at boot time. Therefore, there is a fixed limit on the number of concurrently open files in a system.

The `read()` and `write()` system calls do not take a position in the file as an argument. Rather, the kernel keeps a ***file offset***, which is updated by an appropriate amount after each `read()` or `write()` according to the number of data actually transferred. The offset can be set directly by the `lseek()` system call. If the file descriptor indexed an array of inode pointers instead of file pointers, this offset would have to be kept in the inode. Because more than one process may open the same file, and each such process needs its own offset for the file, keeping the offset in the inode is inappropriate. Thus, the file structure is used to contain the offset. File structures are inherited by the child process after a `fork()`, so several processes may share the same offset location for a file.

The ***inode structure*** pointed to by the file structure is an in-core copy of the inode on the disk. The in-core inode has a few extra fields, such as a reference count of how many file structures are pointing at it, and the file structure has a similar reference count for how many file descriptors refer to it. When a count becomes zero, the entry is no longer needed and may be reclaimed and reused.

### C.7.5 Disk Structures

The file system that the user sees is supported by data on a mass storage device—usually, a disk. The user ordinarily knows of only one file system, but this one logical file system may actually consist of several *physical* file systems, each on a different device. Because device characteristics differ, each separate hardware device defines its own physical file system. In fact, we generally want to partition large physical devices, such as disks, into multiple *logical* devices. Each logical device defines a physical file system. Figure C.9 illustrates how a directory structure is partitioned into file systems, which are mapped onto logical devices, which are partitions of physical devices. The sizes and locations of these partitions were coded into device drivers in earlier systems, but they are maintained on the disk by FreeBSD.

Partitioning a physical device into multiple file systems has several benefits. Different file systems can support different uses. Although most partitions will be used by the file system, at least one will be needed as a swap area for the virtual memory software. Reliability is improved, because software damage is generally limited to only one file system. We can improve efficiency by varying the file-system parameters (such as the block and fragment sizes) for each partition. Also, having separate file systems prevents one program from using all available space for a large file, because files cannot be split across file systems. Finally, disk backups are done per partition, and it is faster to search a backup tape for a file if the partition is smaller. Restoring the full partition from tape is also faster.

The number of file systems on a drive varies according to the size of the disk and the purpose of the computer system as a whole. One file system, the

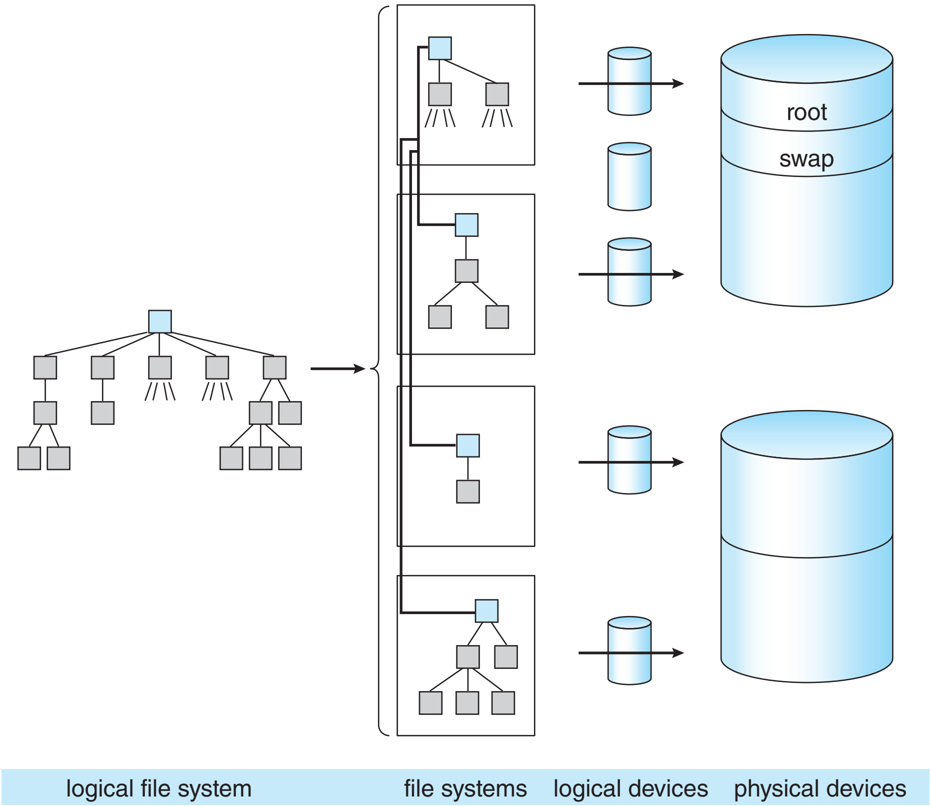


**Figure C.9** Mapping of a logical file system to physical devices.

***root file system***, is always available. Other file systems may be ***mounted***—that is, integrated into the directory hierarchy of the root file system.

A bit in the inode structure indicates that the inode has a file system mounted on it. A reference to this file causes the ***mount table*** to be searched to find the device number of the mounted device. The device number is used to find the inode of the root directory of the mounted file system, and that inode is used. Conversely, if a path-name element is ".." and the directory being searched is the root directory of a file system that is mounted, the mount table is searched to find the inode it is mounted on, and that inode is used.

Each file system is a separate system resource and represents a set of files. The first sector on the logical device is the ***boot block***, possibly containing a primary bootstrap program, which may be used to call a secondary bootstrap program residing in the next 7.5 KB. A system needs only one partition containing boot-block data, but the system manager may install duplicates via privileged programs, to allow booting when the primary copy is damaged. The ***superblock*** contains static parameters of the file system. These parameters include the total size of the file system, the block and fragment sizes of the data blocks, and assorted parameters that affect allocation policies.

### C.7.6 Implementations

The user interface to the file system is simple and well defined, allowing the implementation of the file system itself to be changed without significant effect on the user. The file system was changed between Version 6 and Version 7 of 3BSD, and again between Version 7 and 4BSD. For Version 7, the size of inodes doubled, the maximum file and file-system sizes increased, and the details of free-list handling and superblock information changed. At that time also, `seek()` (with a 16-bit offset) became `lseek()` (with a 32-bit offset), to allow specification of offsets in larger files, but few other changes were visible outside the kernel.

In 4.0BSD, the size of blocks used in the file system was increased from 512 bytes to 1,024 bytes. Although this increased size produced increased internal fragmentation on the disk, it doubled throughput, due mainly to the greater number of data accessed on each disk transfer. This idea was later adopted by System V, along with a number of other ideas, device drivers, and programs.

4.2BSD added the Berkeley Fast File System, which increased speed and was accompanied by new features. Symbolic links required new system calls. Long file names necessitated new directory system calls to traverse the now-complex internal directory structure. Finally, `truncate()` calls were added. The Fast File System was a success and is now found in most implementations of UNIX. Its performance is made possible by its layout and allocation policies, which we discuss next. In Section 14.4.4, we discussed changes made in SunOS to increase disk throughput further.

### C.7.7 Layout and Allocation Policies

The kernel uses a <***logical device number, inode number***> pair to identify a file. The logical device number defines the file system involved. The inodes in the file system are numbered in sequence. In the Version 7 file system, all inodes are in an array immediately following a single superblock at the beginning of the logical device, with the data blocks following the inodes. The inode number is effectively just an index into this array.

With the Version 7 file system, a block of a file can be anywhere on the disk between the end of the inode array and the end of the file system. Free blocks are kept in a linked list in the superblock. Blocks are pushed onto the front of the free list and are removed from the front as needed to serve new files or to extend existing files. Thus, the blocks of a file may be arbitrarily far from both the inode and one another. Furthermore, the more a file system of this kind is used, the more disorganized the blocks in a file become. We can reverse this process only by reinitializing and restoring the entire file system, which is not a convenient task to perform. This process was described in Section 14.7.4.

Another difficulty is that the reliability of the file system is suspect. For speed, the superblock of each mounted file system is kept in memory. Keeping the superblock in memory allows the kernel to access a superblock quickly, especially for using the free list. Every 30 seconds, the superblock is written to the disk, to keep the in-core and disk copies synchronized (by the update program, using the `sync()` system call). However, system bugs or hardware failures may cause a system crash, which destroys the in-core superblock between updates to the disk. Then, the free list on disk does not accurately reflect the state of the disk. To reconstruct it, we must perform a lengthy examination of all blocks in the file system. (This problem remains in the new file system.)

The 4.2BSD file-system implementation is radically different from that of Version 7. This reimplementation was done primarily to improve efficiency and robustness, and most such changes are invisible outside the kernel. Other changes introduced at the same time are visible at both the system-call and the user levels; examples include symbolic links and long file names (up to 255 characters). Most of the changes required for these features were not in the kernel, however, but in the programs that use them.

Space allocation is especially different. The major new concept in FreeBSD is the ***cylinder group***. The cylinder group was introduced to allow localization of the blocks in a file. Each cylinder group occupies one or more consecutive cylinders of the disk, so that disk accesses within the cylinder group require minimal disk head movement. Every cylinder group has a superblock, a cylinder block, an array of inodes, and some data blocks (Figure C.10).

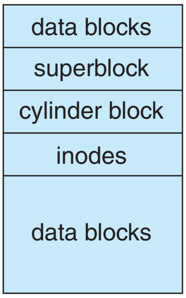


**Figure C.10** 4.3 BSD cylinder group.

The superblocks in all cylinder groups are identical, so that a superblock can be recovered from any one of them in the event of disk corruption. The ***cylinder block*** contains dynamic parameters of the particular cylinder group. These include a bit map of free data blocks and fragments and a bitmap of free inodes. Statistics on recent progress of the allocation strategies are also kept here.

The header information in a cylinder group (the superblock, the cylinder block, and the inodes) is not always at the beginning of the group. If it were, the header information for every cylinder group might be on the same disk platter, and a single disk head crash could wipe out all of them. Therefore, each cylinder group has its header information at a different offset from the beginning of the group.

The directory-listing command `ls` commonly reads all the inodes of every file in a directory, making it desirable for all such inodes to be close together on the disk. For this reason, the inode for a file is usually allocated from the cylinder group containing the inode of the file's parent directory. Not everything can be localized, however, so an inode for a new directory is put in a *different* cylinder group from that of its parent directory. The cylinder group chosen for such a new directory inode is that with the greatest number of unused inodes.

To reduce disk head seeks involved in accessing the data blocks of a file, we allocate blocks from the same cylinder group as often as possible. Because a single file cannot be allowed to take up all the blocks in a cylinder group, a file exceeding a certain size (such as 2 MB) has further block allocation redirected to a different cylinder group; the new group is chosen from among those having more than average free space. If the file continues to grow, allocation is again redirected (at each megabyte) to yet another cylinder group. Thus, all the blocks of a small file are likely to be in the same cylinder group, and the number of long head seeks involved in accessing a large file is kept small.

There are two levels of disk-block-allocation routines. The global policy routines select a desired disk block according to the considerations already discussed. The local policy routines use the specific information recorded in the cylinder blocks to choose a block near the one requested. If the requested block is not in use, it is returned. Otherwise, the routine returns either the block rotationally closest to the one requested in the same cylinder or a block in a different cylinder but in the same cylinder group. If no more blocks are in the cylinder group, a quadratic rehash is done among all the other cylinder groups to find a block. If that fails, an exhaustive search is done. If enough free space (typically 10 percent) is left in the file system, blocks are usually found where desired, the quadratic rehash and exhaustive search are not used, and performance of the file system does not degrade with use.

Because of the increased efficiency of the Fast File System, typical disks are now utilized at 30 percent of their raw transfer capacity. This percentage is a marked improvement over that realized with the Version 7 file system, which used about 3 percent of the bandwidth.

BSD Tahoe introduced the Fat Fast File System, which allows the number of inodes per cylinder group, the number of cylinders per cylinder group, and the number of distinguished rotational positions to be set when the file system is created. FreeBSD previously set these parameters according to the disk hardware type.

## C.8 I/O System

One of the purposes of an operating system is to hide the peculiarities of specific hardware devices from the user. For example, the file system presents a simple, consistent storage facility (the file) independent of the underlying disk hardware. In UNIX, the peculiarities of I/O devices are also hidden from the bulk of the kernel itself by the ***I/O system***. The I/O system consists of a buffer caching system, general device-driver code, and drivers for specific hardware devices. Only the device driver knows the peculiarities of a specific device. The major parts of the I/O system are diagrammed in Figure C.11.

There are three main kinds of I/O in FreeBSD: block devices, character devices, and the socket interface. The socket interface, together with its protocols and network interfaces, will be described in Section C.9.1.

***Block devices*** include disks and tapes. Their distinguishing characteristic is that they are directly addressable in a fixed block size—usually 512 bytes. A block-device driver is required to isolate details of tracks, cylinders, and so on from the rest of the kernel. Block devices are accessible directly through appropriate device special files (such as */dev/rp0*), but they are more commonly accessed indirectly through the file system. In either case, transfers are buffered through the ***block buffer cache***, which has a profound effect on efficiency.

***Character devices*** include terminals and line printers but also include almost everything else (except network interfaces) that does not use the block buffer cache. For instance, */dev/mem* is an interface to physical main memory, and */dev/null* is a bottomless sink for data and an endless source of end-of-file markers. Some devices, such as high-speed graphics interfaces, may have their own buffers or may always do I/O directly into the user's data space; because they do not use the block buffer cache, they are classed as character devices. Terminals and terminal-like devices use ***C-lists***, which are buffers smaller than those of the block buffer cache.

Block devices and character devices are the two main device classes. Device drivers are accessed by one of two arrays of entry points. One array is for block devices; the other is for character devices. A device is distinguished by a class (block or character) and a ***device number***. The device number consists of two parts. The ***major device number*** is used to index the array for character or block devices to find entries into the appropriate device driver. The ***minor***

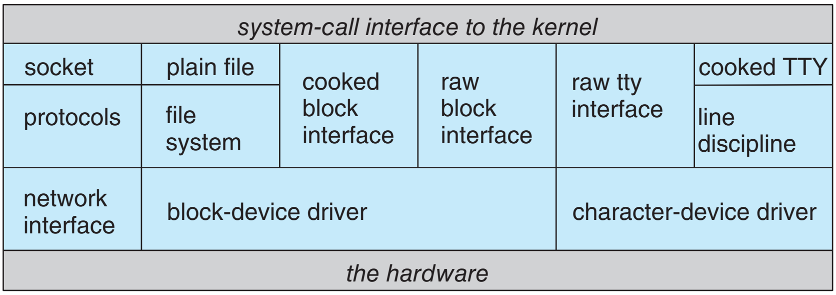


**Figure C.11** 4.3 BSD kernel I/O structure.

***device number*** is interpreted by the device driver as, for example, a logical disk partition or a terminal line.

A device driver is connected to the rest of the kernel only by the entry points recorded in the array for its class and by its use of common buffering systems. This segregation is important for portability and for system configuration.

### C.8.1 Block Buffer Cache

The block devices, as mentioned, use a block buffer cache. The buffer cache consists of a number of buffer headers, each of which can point to a piece of physical memory as well as to a device number and a block number on the device. The buffer headers for blocks not currently in use are kept in several linked lists, one for each of the following:

- Buffers recently used, linked in LRU order (the LRU list)
- Buffers not recently used or without valid contents (the AGE list)
- EMPTY buffers with no physical memory associated with them

The buffers in these lists are also hashed by device and block number for search efficiency.

When a block is wanted from a device (a read), the cache is searched. If the block is found, it is used, and no I/O transfer is necessary. If it is not found, a buffer is chosen from the AGE list or, if that list is empty, the LRU list. Then the device number and block number associated with it are updated, memory is found for it if necessary, and the new data are transferred into it from the device. If there are no empty buffers, the LRU buffer is written to its device (if it is modified), and the buffer is reused.

On a write, if the block in question is already in the buffer cache, the new data are put in the buffer (overwriting any previous data), the buffer header is marked to indicate that the buffer has been modified, and no I/O is immediately necessary. The data will be written when the buffer is needed for other data. If the block is not found in the buffer cache, an empty buffer is chosen (as with a read), and a transfer is done to this buffer. Writes are periodically forced for dirty buffer blocks to minimize potential file-system inconsistencies after a crash.

The number of data in a buffer in FreeBSD is variable, up to a maximum over all file systems, usually 8 KB. The minimum size is the paging-cluster size, usually 1,024 bytes. Buffers are page-cluster aligned, and any page cluster may be mapped into only one buffer at a time, just as any disk block may be mapped into only one buffer at a time. The EMPTY list holds buffer headers, which are used if a physical memory block of 8 KB is split to hold multiple, smaller blocks. Headers are needed for these blocks and are retrieved from EMPTY.

The number of data in a buffer may grow as a user process writes more data following those already in the buffer. When this increase occurs, a new buffer large enough to hold all the data is allocated, and the original data are copied into it, followed by the new data. If a buffer shrinks, a buffer is taken off the empty queue, excess pages are put in it, and that buffer is released to be written to disk.

Some devices, such as magnetic tapes, require that blocks be written in a certain order. Facilities are therefore provided to force a synchronous write of buffers to these devices in the correct order. Directory blocks are also written synchronously, to forestall crash inconsistencies. Consider the chaos that could occur if many changes were made to a directory but the directory entries themselves were not updated!

The size of the buffer cache can have a profound effect on the performance of a system, because, if it is large enough, the percentage of cache hits can be high and the number of actual I/O transfers low. FreeBSD optimizes the buffer cache by continually adjusting the amount of memory used by programs and the disk cache.

Some interesting interactions occur among the buffer cache, the file system, and the disk drivers. When data are written to a disk file, they are buffered in the cache, and the disk driver sorts its output queue according to disk address. These two actions allow the disk driver to minimize disk head seeks and to write data at times optimized for disk rotation. Unless synchronous writes are required, a process writing to disk simply writes into the buffer cache, and the system asynchronously writes the data to disk when convenient. The user process sees very fast writes. When data are read from a disk file, the block I/O system does some read-ahead; however, writes are much nearer to asynchronous than are reads. Thus, output to the disk through the file system is often faster than is input for large transfers, counter to intuition.

### C.8.2 Raw Device Interfaces

Almost every block device also has a character interface, and these are called ***raw device interfaces***. Such an interface differs from the ***block interface*** in that the block buffer cache is bypassed.

Each disk driver maintains a queue of pending transfers. Each record in the queue specifies whether it is a read or a write and gives a main memory address for the transfer (usually in 512-byte increments), a device address for the transfer (usually the address of a disk sector), and a transfer size (in sectors). It is simple to map the information from a block buffer to what is required for this queue.

It is almost as simple to map a piece of main memory corresponding to part of a user process's virtual address space. This mapping is what a raw disk interface, for instance, does. Unbuffered transfers directly to or from a user's virtual address space are thus allowed. The size of the transfer is limited by the physical devices, some of which require an even number of bytes.

The kernel accomplishes transfers for swapping and paging simply by putting the appropriate request on the queue for the appropriate device. No special swapping or paging device driver is needed.

The 4.2BSD file-system implementation was actually written and largely tested as a user process that used a raw disk interface, before the code was moved into the kernel. In an interesting about-face, the Mach operating system has no file system per se. File systems can be implemented as user-level tasks (see Appendix D).

### C.8.3 C-Lists

As mentioned, terminals and terminal-like devices use a character-buffering system that keeps small blocks of characters (usually 28 bytes) in linked lists called C-lists. Although all free character buffers are kept in a single free list, most device drivers that use them limit the number of characters that may be queued at one time for any given terminal line.

There are routines to enqueue and dequeue characters for such lists. A `write()` system call to a terminal enqueues characters on a list for the device. An initial transfer is started, and interrupts cause dequeuing of characters and further transfers.

Input is similarly interrupt driven. Terminal drivers typically support *two* input queues, however, and conversion from the first (raw queue) to the other (canonical queue) is triggered when the interrupt routine puts an end-of-line character on the raw queue. The process doing a read on the device is then awakened, and its system phase does the conversion. The characters thus put on the canonical queue are then available to be returned to the user process by the read.

The device driver can bypass the canonical queue and return characters directly from the raw queue. This mode of operation is known as ***raw mode***. Full-screen editors, as well as other programs that need to react to every keystroke, use this mode.

## C.9 Interprocess Communication

Although many tasks can be accomplished in isolated processes, many others require interprocess communication. Isolated computing systems have long served for many applications, but networking is increasingly important. Furthermore, with the increasing use of personal workstations, resource sharing is becoming more common. Interprocess communication has not traditionally been one of UNIX's strong points.

### C.9.1 Sockets

The pipe (discussed in Section C.4.3) is the IPC mechanism most characteristic of UNIX. A pipe permits a reliable unidirectional byte stream between two processes. It is traditionally implemented as an ordinary file, with a few exceptions. It has no name in the file system, being created instead by the `pipe()` system call. Its size is fixed, and when a process attempts to write to a full pipe, the process is suspended. Once all data previously written into the pipe have been read out, writing continues at the beginning of the file (pipes are not true circular buffers). One benefit of the small size of pipes (usually 4,096 bytes) is that pipe data are seldom actually written to disk; they usually are kept in memory by the normal block buffer cache.

In FreeBSD pipes are implemented as a special case of the ***socket*** mechanism. The socket mechanism provides a general interface not only to facilities such as pipes, which are local to one machine, but also to networking facilities.

Even on the same machine, a pipe can be used only by two processes related through use of the `fork()` system call. The socket mechanism can be used by unrelated processes.

A socket is an endpoint of communication. A socket in use usually has an ***address*** bound to it. The nature of the address depends on the ***communication domain*** of the socket. A characteristic property of a domain is that processes communicating in the same domain use the same ***address format***. A single socket can communicate in only one domain.

The three domains currently implemented in FreeBSD are the UNIX domain (AF_UNIX), the Internet domain (AF_INET), and the XEROX Network Services (NS) domain (AF_NS). The address format of the UNIX domain is that of an ordinary file-system path name, such as */alpha/beta/gamma*. Processes communicating in the Internet domain use DARPA Internet communications protocols (such as TCP/IP) and Internet addresses, which consist of a 32-bit host number and a 32-bit port number (representing a rendezvous point on the host).

There are several **socket types**, which represent classes of services. Each type may or may not be implemented in any communication domain. If a type is implemented in a given domain, it may be implemented by one or more protocols, which may be selected by the user:

- **Stream sockets.** These sockets provide reliable, duplex, sequenced data streams. No data are lost or duplicated in delivery, and there are no record boundaries. This type is supported in the Internet domain by TCP. In the UNIX domain, pipes are implemented as a pair of communicating stream sockets.
- **Sequenced packet sockets.** These sockets provide data streams like those of stream sockets, except that record boundaries are provided. This type is used in the XEROX AF_NS protocol.
- **Datagram sockets.** These sockets transfer messages of variable size in either direction. There is no guarantee that such messages will arrive in the same order they were sent, or that they will be unduplicated, or that they will arrive at all, but the original message (or record) size is preserved in any datagram that does arrive. This type is supported in the Internet domain by UDP.
- **Reliably delivered message sockets.** These sockets transfer messages that are guaranteed to arrive and that otherwise are like the messages transferred using datagram sockets. This type is currently unsupported.
- **Raw sockets.** These sockets allow direct access by processes to the protocols that support the other socket types. The protocols accessible include not only the uppermost ones but also lower-level protocols. For example, in the Internet domain, it is possible to reach TCP, IP beneath that, or an Ethernet protocol beneath that. This capability is useful for developing new protocols.

A set of system calls is specific to the socket facility. The `socket()` system call creates a socket. It takes as arguments specifications of the communication domain, the socket type, and the protocol to be used to support that type. The value returned by the call is a small integer called a **socket descriptor**, which occupies the same name space as file descriptors. The socket descriptor indexes the array of open files in the *u* structure in the kernel and has a file structure allocated for it. The FreeBSD file structure may point to a socket structure instead of to an inode. In this case, certain socket information (such as the socket's type, its message count, and the data in its input and output queues) is kept directly in the socket structure.

For another process to address a socket, the socket must have a name. A name is bound to a socket by the `bind()` system call, which takes the socket descriptor, a pointer to the name, and the length of the name as a byte string. The contents and length of the byte string depend on the address format. The `connect()` system call is used to initiate a connection. The arguments are syntactically the same as those for `bind()`; the socket descriptor represents the local socket, and the address is that of the foreign socket to which the attempt to connect is made.

Many processes that communicate using the socket IPC follow the client–server model. In this model, the *server* process provides a *service* to the *client* process. When the service is available, the server process listens on a well-known address, and the client process uses `connect()` to reach the server.

A server process uses `socket()` to create a socket and `bind()` to bind the well-known address of its service to that socket. Then, it uses the `listen()` system call to tell the kernel that it is ready to accept connections from clients and to specify how many pending connections the kernel should queue until the server can service them. Finally, the server uses the `accept()` system call to accept individual connections. Both `listen()` and `accept()` take as an argument the socket descriptor of the original socket. The system call `accept()` returns a new socket descriptor corresponding to the new connection; the original socket descriptor is still open for further connections. The server usually uses `fork()` to produce a new process after the `accept()` to service the client while the original server process continues to listen for more connections. There are also system calls for setting parameters of a connection and for returning the address of the foreign socket after an `accept()`.

When a connection for a socket type, such as a stream socket, is established, the addresses of both endpoints are known, and no further addressing information is needed to transfer data. The ordinary `read()` and `write()` system calls may then be used to transfer data.

The simplest way to terminate a connection, and to destroy the associated socket, is to use the `close()` system call on its socket descriptor. We may also wish to terminate only one direction of communication of a duplex connection; the `shutdown()` system call can be used for this purpose.

Some socket types, such as datagram sockets, do not support connections. Instead, their sockets exchange datagrams that must be addressed individually. The system calls `sendto()` and `recvfrom()` are used for such connections. Both take as arguments a socket descriptor, a buffer pointer and length, and an address-buffer pointer and length. The address buffer contains the appropriate address for `sendto()` and is filled in with the address of the datagram just received by `recvfrom()`. The number of data actually transferred is returned by both system calls.

The `select()` system call can be used to multiplex data transfers on several file descriptors and/or socket descriptors. It can even be used to allow one server process to listen for client connections for many services and to `fork()`

a process for each connection as the connection is made. The server does a `socket()`, `bind()`, and `listen()` for each service and then does a `select()` on all the socket descriptors. When `select()` indicates activity on a descriptor, the server does an `accept()` on it and forks a process on the new descriptor returned by `accept()`, leaving the parent process to do a `select()` again.

### C.9.2 Network Support

Almost all current UNIX systems support the UUCP network facilities, which are mostly used over dial-up telephone lines to support the UUCP mail network and the USENET news network. These are, however, rudimentary networking facilities; they do not support even remote login, much less remote procedure calls or distributed file systems. These facilities are almost completely implemented as user processes and are not part of the operating system itself.

FreeBSD supports the DARPA Internet protocols UDP, TCP, IP, and ICMP on a wide range of Ethernet, token-ring, and ARPANET interfaces. The framework in the kernel to support these protocols is intended to facilitate the implementation of further protocols, and all protocols are accessible via the socket interface. Rob Gurwitz of BBN wrote the first version of the code as an add-on package for 4.1BSD.

The International Standards Organization's (ISO) Open System Interconnection (OSI) Reference Model for networking prescribes seven layers of network protocols and strict methods of communication between them. An implementation of a protocol may communicate only with a peer entity speaking the same protocol at the same layer or with the protocol–protocol interface of a protocol in the layer immediately above or below in the same system. The ISO networking model is implemented in FreeBSD Reno and 4.4BSD.

The FreeBSD networking implementation, and to a certain extent the socket facility, is more oriented toward the ARPANET Reference Model (ARM). The ARPANET in its original form served as a proof of concept for many networking ideas, such as packet switching and protocol layering. The ARPANET was retired in 1988 because the hardware that supported it was no longer state of the art. Its successors, such as the NSFNET and the Internet, are even larger and serve as communications utilities for researchers and test-beds for Internet gateway research. The ARM predates the ISO model; the ISO model was in large part inspired by the ARPANET research.

Although the ISO model is often interpreted as setting a limit of one protocol communicating per layer, the ARM allows several protocols in the same layer. There are only four protocol layers in the ARM:

- **Process/applications.** This layer subsumes the application, presentation, and session layers of the ISO model. Such user-level programs as the file-transfer protocol (FTP) and Telnet (remote login) exist at this level.

- **Host–host.** This layer corresponds to ISO's transport and the top part of its network layers. Both the Transmission Control Protocol (TCP) and the Internet Protocol (IP) are in this layer, with TCP on top of IP. TCP corresponds to an ISO transport protocol, and IP performs the addressing functions of the ISO network layer.

- **Network interface.** This layer spans the lower part of the ISO network layer and the entire data-link layer. The protocols involved here depend on the physical network type. The ARPANET uses the IMP-Host protocols, whereas an Ethernet uses Ethernet protocols.

- **Network hardware.** The ARM is primarily concerned with software, so there is no explicit network hardware layer. However, any actual network will have hardware corresponding to the ISO physical layer.

The networking framework in FreeBSD is more generalized than either the ISO model or the ARM, although it is most closely related to the ARM (Figure C.12).

User processes communicate with network protocols (and thus with other processes on other machines) via the socket facility. This facility corresponds to the ISO session layer, as it is responsible for setting up and controlling communications.

Sockets are supported by protocols—possibly by several, layered one on another. A protocol may provide services such as reliable delivery, suppression of duplicate transmissions, flow control, and addressing, depending on the socket type being supported and the services required by any higher protocols.

A protocol may communicate with another protocol or with the network interface that is appropriate for the network hardware. There is little restriction in the general framework on what protocols may communicate with what other protocols or on how many protocols may be layered on top of one another. The user process may, by means of the raw socket type, directly access any layer of protocol from the uppermost used to support one of the other socket types, such as streams, down to a raw network interface. This capability is used by routing processes and also for new protocol development.

Most often, there is one **network-interface driver** per network controller type. The network interface is responsible for handling characteristics specific to the local network being addressed. This arrangement ensures that the protocols using the interface do not need to be concerned with these characteristics.

The functions of the network interface depend largely on the **network hardware**, which is whatever is necessary for the network. Some networks may

<table>
<tr><th>ISO reference model</th><th>ARPANET reference model</th><th>4.2BSD layers</th><th>example layering</th></tr>
<tr><td>application</td><td rowspan="3">process applications</td><td rowspan="2">user programs and libraries</td><td rowspan="2">telnet</td></tr>
<tr><td>presentation</td></tr>
<tr><td>session transport</td><td>sockets</td><td>sock_stream</td></tr>
<tr><td></td><td rowspan="2">host–host</td><td rowspan="2">protocol</td><td>TCP</td></tr>
<tr><td rowspan="2">network data link</td><td>IP</td></tr>
<tr><td rowspan="2">network interface</td><td rowspan="2">network interfaces</td><td rowspan="2">Ethernet driver</td></tr>
<tr><td>hardware</td></tr>
<tr><td></td><td>network hardware</td><td>network hardware</td><td>interlan controller</td></tr>
</table>

**Figure C.12** Network reference models and layering.

support reliable transmission at this level, but most do not. Some networks provide broadcast addressing, but many do not.

The socket facility and the networking framework use a common set of memory buffers, or *mbufs*. These are intermediate in size between the large buffers used by the block I/O system and the C-lists used by character devices. An *mbuf* is 128 bytes long; 112 bytes may be used for data, and the rest is used for pointers to link the *mbuf* into queues and for indicators of how much of the data area is actually in use.

Data are ordinarily passed between layers—socket–protocol, protocol–protocol, or protocol–network interface—in *mbufs*. The ability to pass the buffers containing the data eliminates some data copying, but there is still frequently a need to remove or add protocol headers. It is also convenient and efficient for many purposes to be able to hold data that occupy an area the size of the memory-management page. Thus, the data of an *mbuf* may reside not in the *mbuf* itself but elsewhere in memory. There is an *mbuf* page table for this purpose, as well as a pool of pages dedicated to *mbuf* use.

## C.10 Summary

The early advantages of UNIX were that it was written in a high-level language, was distributed in source form, and provided powerful operating-system primitives on an inexpensive platform. These advantages led to UNIX's popularity at educational, research, and government institutions and eventually in the commercial world. This popularity produced many strains of UNIX with varying and improved facilities.

UNIX provides a file system with tree-structured directories. The kernel supports files as unstructured sequences of bytes. Direct access and sequential access are supported through system calls and library routines.

Files are stored as an array of fixed-size data blocks with perhaps a trailing fragment. The data blocks are found by pointers in the inode. Directory entries point to inodes. Disk space is allocated from cylinder groups to minimize head movement and to improve performance.

UNIX is a multiprogrammed system. Processes can easily create new processes with the `fork()` system call. Processes can communicate with pipes or, more generally, sockets. They may be grouped into jobs that may be controlled with signals.

Processes are represented by two structures: the process structure and the user structure. CPU scheduling is a priority algorithm with dynamically computed priorities that reduces to round-robin scheduling in the extreme case.

FreeBSD memory management uses swapping supported by paging. A `pagedaemon` process uses a modified second-chance page-replacement algorithm to keep enough free frames to support the executing processes.

Page and file I/O uses a block buffer cache to minimize the amount of actual I/O. Terminal devices use a separate character-buffering system.

Networking support is one of the most important features in FreeBSD. The socket concept provides the programming mechanism to access other processes, even across a network. Sockets provide an interface to several sets of protocols.

## Further Reading

[McKusick et al. (2015)] provides a good general discussion of FreeBSD. A modern scheduler for FreeBSD is described in [Roberson (2003)]. Locking in the Multithreaded FreeBSD Kernel is described in [Baldwin (2002)].

FreeBSD is described in *The FreeBSD Handbook*, which can be downloaded from http://www.freebsd.org.

## Bibliography

**[Baldwin (2002)]** J. Baldwin, "Locking in the Multithreaded FreeBSD Kernel", *USENIX BSD* (2002).

**[McKusick et al. (2015)]** M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson, *The Design and Implementation of the FreeBSD UNIX Operating System–Second Edition*, Pearson (2015).

**[Roberson (2003)]** J. Roberson, "ULE: A Modern Scheduler For FreeBSD", *Proceedings of the USENIX BSDCon Conference* (2003), pages 17–28.

# The Mach System

## Appendix D

**This chapter was first written in 1991 and has been updated over time but is no longer modified**

In this appendix we examine the Mach operating system. Mach is designed to incorporate the many recent innovations in operating-system research to produce a fully functional, technically advanced system. Unlike UNIX, which was developed without regard for multiprocessing, Mach incorporates multiprocessing support throughout. This support is exceedingly flexible, accommodating shared-memory systems as well as systems with no memory shared between processors. Mach is designed to run on computer systems ranging from one processor to thousands of processors. In addition, it is easily ported to many varied computer architectures. A key goal of Mach is to be a distributed system capable of functioning on heterogeneous hardware.

Although many experimental operating systems are being designed, built, and used, Mach satisfies the needs of most users better than the others because it offers full compatibility with UNIX 4.3 BSD. This compatibility also gives us a unique opportunity to compare two functionally similar, but internally dissimilar, operating systems. Mach and UNIX differ in their emphases, so our Mach discussion does not exactly parallel our UNIX discussion. In addition, we do not include a section on the user interface, because that component is similar to the user interface in 4.3 BSD. As you will see, Mach provides the ability to layer emulation of other operating systems as well; other operating systems can even run concurrently with Mach.

## D.1 History of the Mach System

Mach traces its ancestry to the Accent operating system developed at Carnegie Mellon University (CMU). Although Accent pioneered a number of novel operating-system concepts, its utility was limited by its inability to execute UNIX applications and its strong ties to a single hardware architecture, which made it difficult to port. Mach's communication system and philosophy are derived from Accent, but many other significant portions of the system (for example, the virtual memory system and the management of tasks and threads) were developed from scratch. An important goal of the Mach effort was support for multiprocessors.

Mach's development followed an evolutionary path from BSD UNIX systems. Mach code was initially developed inside the 4.2BSD kernel, with BSD kernel components replaced by Mach components as the Mach components were completed. The BSD components were updated to 4.3 BSD when that became available. By 1986, the virtual memory and communication subsystems were running on the DEC VAX computer family, including multiprocessor versions of the VAX. Versions for the IBM RT/PC and for Sun 3 workstations followed shortly; 1987 saw the completion of the Encore Multimax and Sequent Balance multiprocessor versions, including task and thread support, as well as the first official releases of the system, Release 0 and Release 1.

Through Release 2, Mach provided compatibility with the corresponding BSD systems by including much of BSD's code in the kernel. The new features and capabilities of Mach made the kernels in these releases larger than the corresponding BSD kernels. Mach 3 (Figure D.1) moved the BSD code outside of the kernel, leaving a much smaller microkernel. This system implements only basic Mach features in the kernel; all UNIX-specific code has been evicted to run in user-mode servers. Excluding UNIX-specific code from the kernel allows replacement of BSD with another operating system or the simultaneous execution of multiple operating-system interfaces on top of the microkernel. In addition to BSD, user-mode implementations have been developed for DOS, the Macintosh operating system, and OSF/1. This approach has similarities to the virtual machine concept, but the virtual machine is defined by software (the Mach kernel interface), rather than by hardware. With Release 3.0, Mach became available on a wide variety of systems, including single-processor Sun, Intel, IBM, and DEC machines and multiprocessor DEC, Sequent, and Encore systems.

Mach was propelled to the forefront of industry attention when the Open Software Foundation (OSF) announced in 1989 that it would use Mach 2.5 as the basis for its new operating system, OSF/1. The release of OSF/1 occurred a year later, and it now competes with UNIX System V, Release 4, the operating system of choice among ***UNIX International (UI)*** members. OSF members include key technological companies such as IBM, DEC, and HP. Mach 2.5 is also the basis for the operating system on the NeXT workstation, the brainchild of Steve Jobs, of Apple Computer fame. OSF is evaluating Mach 3 as the basis for a future

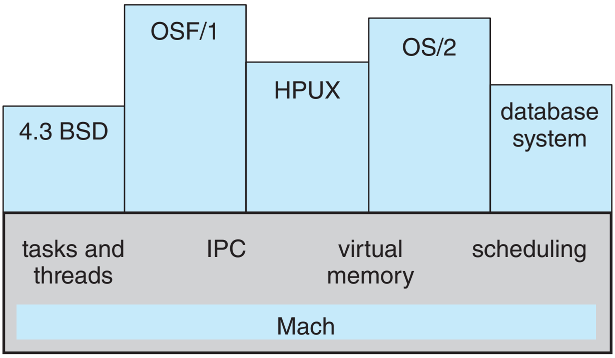


**Figure D.1** Mach 3 structure.

operating-system release, and research on Mach continues at CMU, OSF, and elsewhere.

## D.2 Design Principles

The Mach operating system was designed to provide basic mechanisms that most current operating systems lack. The goal is to design an operating system that is BSD-compatible and, in addition, excels in the following areas:

- Support for diverse architectures, including multiprocessors with varying degrees of shared memory access: uniform memory access (UMA), non-uniform memory access (NUMA), and no remote memory access (NORMA)
- Ability to function with varying intercomputer network speeds, from wide-area networks to high-speed local-area networks and tightly coupled multiprocessors
- Simplified kernel structure, with a small number of abstractions (in turn, these abstractions are sufficiently general to allow other operating systems to be implemented on top of Mach.)
- Distributed operation, providing network transparency to clients and an object-oriented organization both internally and externally
- Integrated memory management and interprocess communication, to provide efficient communication of large numbers of data as well as communication-based memory management
- Heterogeneous system support, to make Mach widely available and interoperable among computer systems from multiple vendors

The designers of Mach have been heavily influenced by BSD (and by UNIX in general), whose benefits include

- A simple programmer interface, with a good set of primitives and a consistent set of interfaces to system facilities
- Easy portability to a wide class of single processors
- An extensive library of utilities and applications
- The ability to combine utilities easily via pipes

Of course, the designers also wanted to redress what they saw as the drawbacks of BSD:

- A kernel that has become the repository of many redundant features—and that consequently is difficult to manage and modify
- Original design goals that made it difficult to provide support for multiprocessors, distributed systems, and shared program libraries (for instance, because the kernel was designed for single processors, it has no provisions for locking code or data that other processors might be using.)

- Too many fundamental abstractions, providing too many similar, competing means with which to accomplish the same tasks

The development of Mach continues to be a huge undertaking. The benefits of such a system are equally large, however. The operating system runs on many existing single-processor and multiprocessor architectures, and it can be easily ported to future ones. It makes research easier, because computer scientists can add features via user-level code, instead of having to write their own tailor-made operating system. Areas of experimentation include operating systems, databases, reliable distributed systems, multiprocessor languages, security, and distributed artificial intelligence. In its current version, the Mach system is usually as efficient as other major versions of UNIX when performing similar tasks.

## D.3 System Components

To achieve the design goals of Mach, the developers reduced the operating-system functionality to a small set of basic abstractions, out of which all other functionality can be derived. The Mach approach is to place as little as possible within the kernel but to make what is there powerful enough that all other features can be implemented at the user level.

Mach's design philosophy is to have a simple, extensible kernel, concentrating on communication facilities. For instance, all requests to the kernel, and all data movement among processes, are handled through one communication mechanism. Mach is therefore able to provide system-wide protection to its users by protecting the communication mechanism. Optimizing this one communication path can result in significant performance gains, and it is simpler than trying to optimize several paths. Mach is extensible, because many traditionally kernel-based functions can be implemented as user-level servers. For instance, all pagers (including the default pager) can be implemented externally and called by the kernel for the user.

Mach is an example of an object-oriented system where the data and the operations that manipulate that data are encapsulated into an abstract object. Only the operations of the object are able to act on the entities defined in it. The details of how these operations are implemented are hidden, as are the internal data structures. Thus, a programmer can use an object only by invoking its defined, exported operations. A programmer can change the internal operations without changing the interface definition, so changes and optimizations do not affect other aspects of system operation. The object-oriented approach supported by Mach allows objects to reside anywhere in a network of Mach systems, transparent to the user. The `port` mechanism, discussed later in this section, makes all of this possible.

Mach's primitive abstractions are the heart of the system and are as follows:

- A **task** is an execution environment that provides the basic unit of resource allocation. It consists of a virtual address space and protected access to system resources via ports, and it may contain one or more threads.
- A **thread** is the basic unit of execution and must run in the context of a task (which provides the address space). All threads within a task share the

task's resources (ports, memory, and so on). There is no notion of a ***process*** in Mach. Rather, a traditional process is implemented as a task with a single thread of control.

- A **port** is the basic object-reference mechanism in Mach and is implemented as a kernel-protected communication channel. Communication is accomplished by sending messages to ports; messages are queued at the destination port if no thread is immediately ready to receive them. Ports are protected by kernel-managed capabilities, or **port rights**. A task must have a port right to send a message to a port. The programmer invokes an operation on an object by ***sending*** a message to a port associated with that object. The object being represented by a port ***receives*** the messages.
- A **port set** is a group of ports sharing a common message queue. A thread can receive messages for a port set and thus service multiple ports. Each received message identifies the individual port (within the set) from which it was received. The receiver can use this to identify the object referred to by the message.
- A **message** is the basic method of communication between threads in Mach. It is a typed collection of data objects. For each object, it may contain the actual data or a pointer to out-of-line data. Port rights are passed in messages; this is the only way to move them among tasks. (Passing a port right in shared memory does not work, because the Mach kernel will not permit the new task to use a right obtained in this manner.)
- A **memory object** is a source of memory. Tasks can access it by mapping portions of an object (or the entire object) into their address spaces. The object can be managed by a user-mode external memory manager. One example is a file managed by a file server; however, a memory object can be any object for which memory-mapped access makes sense. A mapped buffer implementation of a UNIX pipe is another example.

Figure D.2 illustrates these abstractions, which we explain further in the remainder of this chapter.

An unusual feature of Mach, and a key to the system's efficiency, is the blending of memory and interprocess-communication (IPC) features. Whereas some other distributed systems (such as Solaris, with its NFS features) have special-purpose extensions to the file system to extend it over a network, Mach provides a general-purpose, extensible merger of memory and messages at the heart of its kernel. This feature not only allows Mach to be used for distributed and parallel programming but also helps in the implementation of the kernel itself.

Mach connects memory management and IPC by allowing each to be used in the implementation of the other. Memory management is based on the use of memory objects. A memory object is represented by a port (or ports), and IPC messages are sent to this port to request operations (for example, `pagein`, `pageout`) on the object. Because IPC is used, memory objects can reside on remote systems and be accessed transparently. The kernel caches the contents of memory objects in local memory. Conversely, memory-management techniques are used in the implementation of message passing. Where possible,

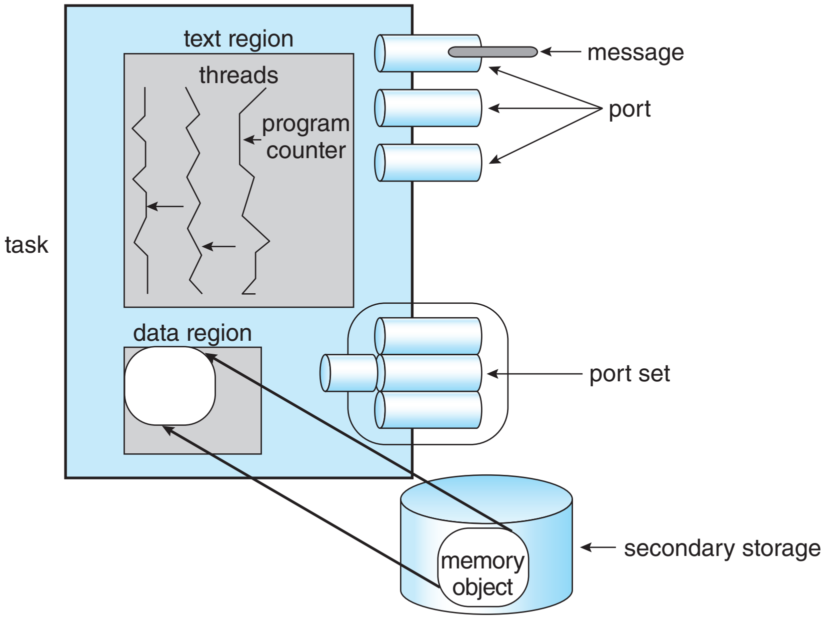


**Figure D.2** Mach's basic abstractions.

Mach passes messages by moving pointers to shared memory objects, rather than by copying the objects themselves.

IPC tends to involve considerable system overhead. For intrasystem messages, it is generally less efficient than communication accomplished through shared memory. Because Mach is a message-based kernel, message handling must be carried out efficiently. Most of the inefficiency of message handling in traditional operating systems is due to either the copying of messages from one task to another (for intracomputer messages) or low network-transfer speed (for intercomputer messages). To solve these problems, Mach uses virtual memory remapping to transfer the contents of large messages. In other words, the message transfer modifies the receiving task's address space to include a copy of the message contents. Virtual copy (or copy-on-write) techniques are used to avoid or delay the actual copying of the data. This approach has several advantages:

- Increased flexibility in memory management for user programs
- Greater generality, allowing the virtual copy approach to be used in tightly and loosely coupled computers
- Improved performance over UNIX message passing
- Easier task migration (because ports are location independent, a task and all its ports can be moved from one machine to another. All tasks that previously communicated with the moved task can continue to do so because they reference the task only by its ports and communicate via messages to these ports.)

In the sections that follow, we detail the operation of process management, IPC, and memory management. Then, we discuss Mach's chameleonlike ability to support multiple operating-system interfaces.

## D.4 Process Management

A task can be thought of as a traditional process that does not have an instruction pointer or a register set. A task contains a virtual address space, a set of port rights, and accounting information. A task is a passive entity that does nothing unless it has one or more threads executing in it.

### D.4.1 Basic Structure

A task containing one thread is similar to a UNIX process. Just as a `fork()` system call produces a new UNIX process, Mach creates a new task by using `fork()`. The new task's memory is a duplicate of the parent's address space, as dictated by the **inheritance attributes** of the parent's memory. The new task contains one thread, which is started at the same point as the creating `fork()` call in the parent. Threads and tasks can also be suspended and resumed.

Threads are especially useful in server applications, which are common in UNIX, since one task can have multiple threads to service multiple requests to the task. Threads also allow efficient use of parallel computing resources. Rather than having one process on each processor, with the corresponding performance penalty and operating-system overhead, a task can have its threads spread among parallel processors. Threads add efficiency to user-level programs as well. For instance, in UNIX, an entire process must wait when a page fault occurs or when a system call is executed. In a task with multiple threads, only the thread that causes the page fault or executes a system call is delayed; all other threads continue executing. Because Mach has kernel-supported threads (Chapter 4), the threads have some cost associated with them. They must have supporting data structures in the kernel, and more complex kernel-scheduling algorithms must be provided. These algorithms and thread states are discussed in Chapter 4.

At the user level, threads may be in one of two states:

- **Running.** The thread is either executing or waiting to be allocated a processor. A thread is considered to be running even if it is blocked within the kernel (waiting for a page fault to be satisfied, for instance).
- **Suspended.** The thread is neither executing on a processor nor waiting to be allocated a processor. A thread can resume its execution only if it is returned to the running state.

These two states are also associated with a task. An operation on a task affects all threads in a task, so suspending a task involves suspending all the threads in it. Task and thread suspensions are separate, independent mechanisms, however, so resuming a thread in a suspended task does not resume the task.

Mach provides primitives from which thread-synchronization tools can be built. This practice is consistent with Mach's philosophy of providing minimum yet sufficient functionality in the kernel. The Mach IPC facility can be used for synchronization, with processes exchanging messages at rendezvous points. Thread-level synchronization is provided by calls to start and stop threads at appropriate times. A ***suspend count*** is kept for each thread. This count allows multiple `suspend()` calls to be executed on a thread, and only when an equal number of `resume()` calls occur is the thread resumed. Unfortunately, this feature has its own limitation. Because it is an error for a `start()` call to be executed before a `stop()` call (the suspend count would become negative), these routines cannot be used to synchronize shared data access. However, `wait()` and `signal()` operations associated with semaphores, and used for synchronization, can be implemented via the IPC calls. We discuss this method in Section D.5.

### D.4.2 The C Threads Package

Mach provides low-level but flexible routines instead of polished, large, and more restrictive functions. Rather than making programmers work at this low level, Mach provides many higher-level interfaces for programming in C and other languages. For instance, the C threads package provides multiple threads of control, shared variables, mutual exclusion for critical sections, and condition variables for synchronization. In fact, C threads is one of the major influences on the POSIX Pthreads standard, which many operating systems support. As a result, there are strong similarities between the C threads and Pthreads programming interfaces. The thread-control routines include calls to perform these tasks:

- Create a new thread within a task, given a function to execute and parameters as input. The thread then executes concurrently with the creating thread, which receives a thread identifier when the call returns.
- Destroy the calling thread, and return a value to the creating thread.
- Wait for a specific thread to terminate before allowing the calling thread to continue. This call is a synchronization tool, much like the UNIX `wait()` system calls.
- Yield use of a processor, signaling that the scheduler can run another thread at this point. This call is also useful in the presence of a preemptive scheduler, as it can be used to relinquish the CPU voluntarily before the time quantum (or scheduling interval) expires if a thread has no use for the CPU.

Mutual exclusion is achieved through the use of spinlocks, as discussed in Chapter 6. The routines associated with mutual exclusion are these:

- The routine `mutex_alloc()` dynamically creates a mutex variable.
- The routine `mutex_free()` deallocates a dynamically created mutex variable.
- The routine `mutex_lock()` locks a mutex variable. The executing thread loops in a spinlock until the lock is attained. A deadlock results if a thread with a lock tries to lock the same mutex variable. Bounded waiting is

not guaranteed by the C threads package. Rather, it is dependent on the hardware instructions used to implement the mutex routines.

- The routine `mutex_unlock()` unlocks a mutex variable, much like the typical `signal()` operation of a semaphore.

General synchronization without busy waiting can be achieved through the use of condition variables, which can be used to implement a monitor, as described in Chapter 6. A condition variable is associated with a mutex variable and reflects a Boolean state of that variable. The routines associated with general synchronization are these:

- The routine `condition_alloc()` dynamically allocates a condition variable.
- The routine `condition_free()` deletes a dynamically created condition variable allocated as a result of `condition_alloc()`.
- The routine `condition_wait()` unlocks the associated mutex variable and blocks the thread until a `condition_signal()` is executed on the condition variable, indicating that the event being waited for may have occurred. The mutex variable is then locked, and the thread continues. A `condition_signal()` does not guarantee that the condition still holds when the unblocked thread finally returns from its `condition_wait()` call, so the awakened thread must loop, executing the `condition_wait()` routine until it is unblocked and the condition holds.

As an example of the C threads routines, consider the bounded-buffer synchronization problem of Section 7.1.1. The producer and consumer are represented as threads that access the common bounded-buffer pool. We use a mutex variable to protect the buffer while it is being updated. Once we have exclusive access to the buffer, we use condition variables to block the producer thread if the buffer is full and to block the consumer thread if the buffer is empty. As in Chapter 6, we assume that the buffer consists of `n` slots, each capable of holding one item. The `mutex` semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The `empty` and `full` semaphores count the number of empty and full buffers, respectively. The semaphore `empty` is initialized to the value `n`; the semaphore `full` is initialized to the value 0. The condition variable `nonempty` is true while the buffer has items in it, and `nonfull` is true if the buffer has an empty slot. The first step includes the allocation of the mutex and condition variables:

```
mutex_alloc(mutex);
condition_alloc(nonempty, nonfull);
```

The code for the producer thread is shown in Figure D.3, and the code for the consumer thread is shown in Figure D.4. When the program terminates, the mutex and condition variables need to be deallocated:

```
mutex_free(mutex);
condition_free(nonempty, nonfull);
```

```
do {
      . . .
   // produce an item into nextp
      . . .
   mutex_lock(mutex);
   while(full)
      condition_wait(nonfull, mutex);
      . . .
   // add nextp to buffer
      . . .
   condition_signal(nonempty);
   mutex_unlock(mutex);
} while(TRUE);
```

**Figure D.3** The structure of the producer process.

### D.4.3 The CPU Scheduler

The CPU scheduler for a thread-based multiprocessor operating system is more complex than its process-based relatives. There are generally more threads in a multithreaded system than there are processes in a multitasking system. Keeping track of multiple processors is also difficult and is a relatively new area of research. Mach uses a simple policy to keep the scheduler manageable. Only threads are scheduled, so no knowledge of tasks is needed in the scheduler. All threads compete equally for resources, including time quanta.

Each thread has an associated priority number ranging from 0 through 127, which is based on the exponential average of its usage of the CPU. That is, a thread that recently used the CPU for a large amount of time has the lowest

```
do {
   mutex_lock(mutex);
   while(empty)
      condition_wait(nonempty, mutex);
      . . .
   // remove an item from the buffe to nextc
      . . .
   condition_signal(nonfull);
   mutex_unlock(mutex);
      . . .
   // consume the item in nextc
      . . .
} until(FALSE);
```

**Figure D.4** The structure of the consumer process.

priority. Mach uses the priority to place the thread in one of 32 global run queues. These queues are searched in priority order for waiting threads when a processor becomes idle. Mach also keeps per-processor, or local, run queues. A local run queue is used for threads that are bound to an individual processor. For instance, a device driver for a device connected to an individual CPU must run on only that CPU.

Instead of a central dispatcher that assigns threads to processors, each processor consults the local and global run queues to select the appropriate next thread to run. Threads in the local run queue have absolute priority over those in the global queues, because it is assumed that they are performing some chore for the kernel. The run queues—like most other objects in Mach—are locked when they are modified to avoid simultaneous changes by multiple processors. To speed dispatching of threads on the global run queue, Mach maintains a list of idle processors.

Additional scheduling difficulties arise from the multiprocessor nature of Mach. A fixed time quantum is not appropriate because, for instance, there may be fewer runnable threads than there are available processors. It would be wasteful to interrupt a thread with a context switch to the kernel when that thread's quantum runs out, only to have the thread be placed right back in the running state. Thus, instead of using a fixed-length quantum, Mach varies the size of the time quantum inversely with the total number of threads in the system. It keeps the time quantum over the entire system constant, however. For example, in a system with 10 processors, 11 threads, and a 100-millisecond quantum, a context switch needs to occur on each processor only once per second to maintain the desired quantum.

Of course, complications still exist. Even relinquishing the CPU while waiting for a resource is more difficult than it is on traditional operating systems. First, a thread must issue a call to alert the scheduler that the thread is about to block. This alert avoids race conditions and deadlocks, which could occur when the execution takes place in a multiprocessor environment. A second call actually causes the thread to be moved off the run queue until the appropriate event occurs. The scheduler uses many other internal thread states to control thread execution.

### D.4.4 Exception Handling

Mach was designed to provide a single, simple, consistent exception-handling system, with support for standard as well as user-defined exceptions. To avoid redundancy in the kernel, Mach uses kernel primitives whenever possible. For instance, an exception handler is just another thread in the task in which the exception occurs. Remote procedure call (RPC) messages are used to synchronize the execution of the thread causing the exception (the ***victim***) and that of the handler and to communicate information about the exception between the victim and handler. Mach exceptions are also used to emulate the BSD `signal` package.

Disruptions to normal program execution come in two varieties: internally generated exceptions and external interrupts. Interrupts are asynchronously generated disruptions of a thread or task, whereas exceptions are caused by the occurrence of unusual conditions during a thread's execution. Mach's general-purpose exception facility is used for error detection and debugger support.

This facility is also useful for other functions, such as taking a core dump of a bad task, allowing tasks to handle their own errors (mostly arithmetic), and emulating instructions not implemented in hardware.

Mach supports two different granularities of exception handling. Error handling is supported by per-thread exception handling, whereas debuggers use per-task handling. It makes little sense to try to debug only one thread or to have exceptions from multiple threads invoke multiple debuggers. Aside from this distinction, the only difference between the two types of exceptions lies in their inheritance from a parent task. Task-wide exception-handling facilities are passed from the parent to child tasks, so debuggers are able to manipulate an entire tree of tasks. Error handlers are not inherited and default to no handler at thread- and task-creation time. Finally, error handlers take precedence over debuggers if the exceptions occur simultaneously. The reason for this approach is that error handlers are normally part of the task and therefore should execute normally even in the presence of a debugger.

Exception handling proceeds as follows:

- The victim thread causes notification of an exception's occurrence via a `raise()` RPC message sent to the handler.
- The victim then calls a routine to wait until the exception is handled.
- The handler receives notification of the exception, usually including information about the exception, the thread, and the task causing the exception.
- The handler performs its function according to the type of exception. The handler's action involves ***clearing*** the exception, causing the victim to ***resume***, or ***terminating*** the victim thread.

To support the execution of BSD programs, Mach needs to support BSD-style signals. Signals provide software-generated interrupts and exceptions. Unfortunately, signals are of limited functionality in multithreaded operating systems. The first problem is that, in UNIX, a signal's handler must be a routine in the process receiving the signal. If the signal is caused by a problem in the process itself (for example, a division by 0), the problem cannot be remedied, because a process has limited access to its own context. A second, more troublesome aspect of signals is that they were designed for only single-threaded programs. For instance, it makes no sense for all threads in a task to get a signal, but how can a signal be seen by only one thread?

Because the signal system must work correctly with multithreaded applications for Mach to run 4.3 BSD programs, signals could not be abandoned. Producing a functionally correct signal package required several rewrites of the code, however. A final problem with UNIX signals is that they can be lost. This loss occurs when another signal of the same type occurs before the first is handled. Mach exceptions are queued as a result of their RPC implementation.

Externally generated signals, including those sent from one BSD process to another, are processed by the BSD server section of the Mach 2.5 kernel. Their behavior is therefore the same as it is under BSD. Hardware exceptions are a different matter, because BSD programs expect to receive hardware exceptions as signals. Therefore, a hardware exception caused by a thread must arrive at the thread as a signal. So that this result is produced, hardware exceptions are converted to exception RPCs. For tasks and threads that do not make explicit use of the Mach exception-handling facility, the destination of this RPC defaults to an in-kernel task. This task has only one purpose: Its thread runs in a continuous loop, receiving the exception RPCs. For each RPC, it converts the exception into the appropriate signal, which is sent to the thread that caused the hardware exception. It then completes the RPC, clearing the original exception condition. With the completion of the RPC, the initiating thread reenters the run state. It immediately sees the signal and executes its signal-handling code. In this manner, all hardware exceptions begin in a uniform way—as exception RPCs. Threads not designed to handle such exceptions, however, receive the exceptions as they would on a standard BSD system—as signals. In Mach 3.0, the signal-handling code is moved entirely into a server, but the overall structure and flow of control is similar to those of Mach 2.5.

## D.5 Interprocess Communication

Most commercial operating systems, such as UNIX, provide communication between processes and between hosts with fixed, global names (or Internet addresses). There is no location independence of facilities, because any remote system needing to use a facility must know the name of the system providing that facility. Usually, data in the messages are untyped streams of bytes. Mach simplifies this picture by sending messages between location-independent ports. The messages contain typed data for ease of interpretation. All BSD communication methods can be implemented with this simplified system.

The two components of Mach IPC are ports and messages. Almost everything in Mach is an object, and all objects are addressed via their communication ports. Messages are sent to these ports to initiate operations on the objects by the routines that implement the objects. By depending on only ports and messages for all communication, Mach delivers location independence of objects and security of communication. Data independence is provided by the NetMsgServer task (Section D.5.3).

Mach ensures security by requiring that message senders and receivers have ***rights***. A right consists of a port name and a capability—send or receive—on that port, and is much like a capability in object-oriented systems. Only one task may have receive rights to any given port, but many tasks may have send rights. When an object is created, its creator also allocates a port to represent the object and obtains the access rights to that port. Rights can be given out by the creator of the object, including the kernel, and are passed in messages. If the holder of a receive right sends that right in a message, the receiver of the message gains the right, and the sender loses it. A task may allocate ports to allow access to any objects it owns or for communication. The destruction of either a port or the holder of the receive right causes the revocation of all rights to that port, and the tasks holding send rights can be notified if desired.

### D.5.1 Ports

A port is implemented as a protected, bounded queue within the kernel of the system on which the object resides. If a queue is full, a sender may abort the send, wait for a slot to become available in the queue, or have the kernel deliver the message.

Several system calls provide the port with the following functionalities:

- Allocate a new port in a specified task and give the caller's task all access rights to the new port. The port name is returned.
- Deallocate a task's access rights to a port. If the task holds the receive right, the port is destroyed, and all other tasks with send rights are, potentially, notified.
- Get the current status of a task's port.
- Create a backup port, which is given the receive right for a port if the task containing the receive right requests its deallocation or terminates.

When a task is created, the kernel creates several ports for it. The function `task_self()` returns the name of the port that represents the task in calls to the kernel. For instance, to allocate a new port, a task calls `port_allocate()` with `task_self()` as the name of the task that will own the port. Thread creation results in a similar `thread_self()` thread kernel port. This scheme is similar to the standard process-ID concept found in UNIX. Another port is returned by `task_notify()`; this is the port to which the kernel will send event-notification messages (such as notifications of port terminations).

Ports can also be collected into ***port sets***. This facility is useful if one thread is to service requests coming in on multiple ports—for example, for multiple objects. A port may be a member of no more than one port set at a time. Furthermore, if a port is in a set, it may not be used directly to receive messages. Instead, messages will be routed to the port set's queue. A port set may not be passed in messages, unlike a port. Port sets are objects that serve a purpose similar to the 4.3 BSD `select()` system call, but they are more efficient.

### D.5.2 Messages

A message consists of a fixed-length header and a variable number of typed data objects. The header contains the destination's port name, the name of the reply port to which return messages should be sent, and the length of the message (Figure D.5). The data in the message (or in-line data) were limited to less than 8 KB in Mach 2.5 systems, but Mach 3.0 has no limit. Each data section may be a simple type (numbers or characters), port rights, or pointers to out-of-line data. Each section has an associated type, so that the receiver can unpack the data correctly even if it uses a byte ordering different from that used by the sender. The kernel also inspects the message for certain types of data. For instance, the kernel must process port information within a message, either by translating the port name into an internal port data structure address or by forwarding it for processing to the NetMsgServer (Section D.5.3).

The use of pointers in a message provides the means to transfer the entire address space of a task in one single message. The kernel also must process pointers to out-of-line data, since a pointer to data in the sender's address space would be invalid in the receiver's—especially if the sender and receiver reside on different systems. Generally, systems send messages by copying the data from the sender to the receiver. Because this technique can be inefficient, especially for large messages, Mach takes a different approach. The data refer-

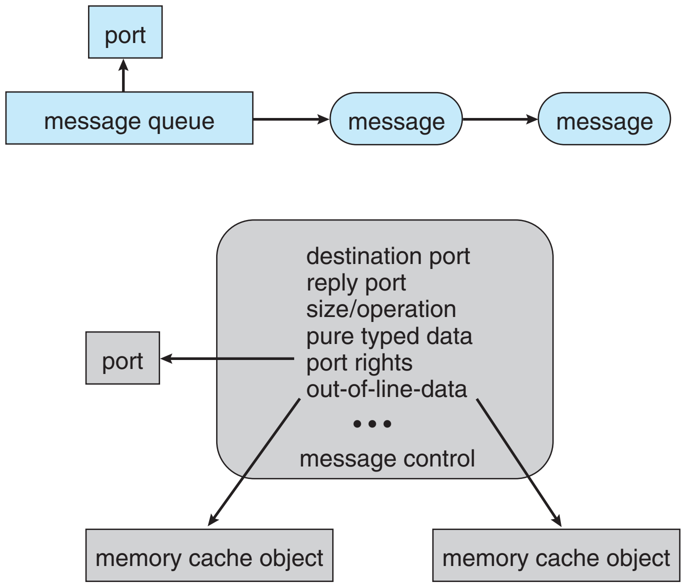


**Figure D.5** Mach messages.

enced by a pointer in a message being sent to a port on the same system are not copied between the sender and the receiver. Instead, the address map of the receiving task is modified to include a copy-on-write copy of the pages of the message. This operation is *much* faster than a data copy and makes message passing more efficient. In essence, message passing is implemented via virtual memory management.

In Version 2.5, this operation was implemented in two phases. A pointer to a region of memory caused the kernel to map that region into its own space temporarily, setting the sender's memory map to copy-on-write mode to ensure that any modifications did not affect the original version of the data. When a message was received at its destination, the kernel moved its mapping to the receiver's address space, using a newly allocated region of virtual memory within that task.

In Version 3, this process was simplified. The kernel creates a data structure that would be a copy of the region if it were part of an address map. On receipt, this data structure is added to the receiver's map and becomes a copy accessible to the receiver.

The newly allocated regions in a task do not need to be contiguous with previous allocations, so Mach virtual memory is said to be ***sparse***, consisting of regions of data separated by unallocated addresses. A full message transfer is shown in Figure D.6.

### D.5.3 The NetMsgServer

For a message to be sent between computers, the message's destination must be located, and the message must be transmitted to the destination. UNIX traditionally leaves these mechanisms to the low-level network protocols, which require the use of statically assigned communication endpoints (for example,

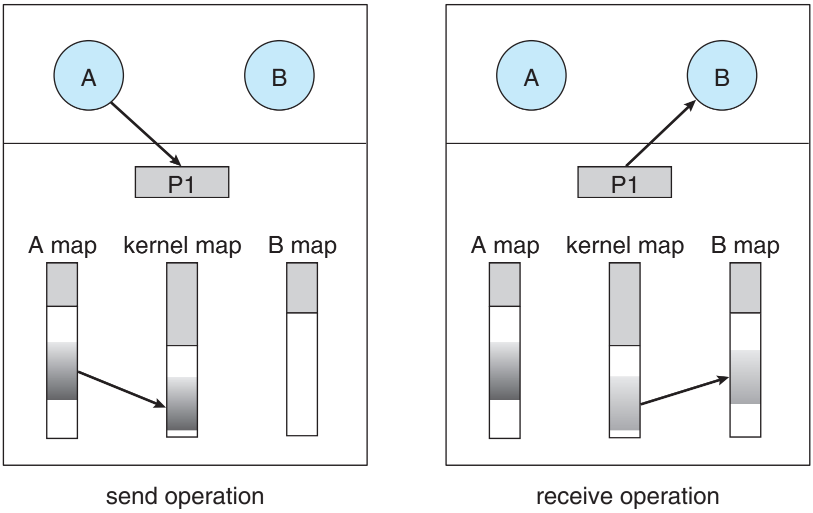


**Figure D.6** Mach message transfer.

the port number for services based on TCP or UDP). One of Mach's tenets is that all objects within the system are location independent and that the location is transparent to the user. This tenet requires Mach to provide location-transparent naming and transport to extend IPC across multiple computers.

This naming and transport are performed by the **Network Message Server** (**NetMsgServer**), a user-level, capability-based networking daemon that forwards messages between hosts. It also provides a primitive network-wide name service that allows tasks to register ports for lookup by tasks on any other computer in the network. Mach ports can be transferred only in messages, and messages must be sent to ports. The primitive name service solves the problem of transferring the first port. Subsequent IPC interactions are fully transparent, because the NetMsgServer tracks all rights and out-of-line memory passed in intercomputer messages and arranges for the appropriate transfers. The NetMsgServers maintain among themselves a distributed database of port rights that have been transferred between computers and of the ports to which these rights correspond.

The kernel uses the NetMsgServer when a message needs to be sent to a port that is not on the kernel's computer. Mach's kernel IPC is used to transfer the message to the local NetMsgServer. The NetMsgServer then uses whatever network protocols are appropriate to transfer the message to its peer on the other computer. The notion of a NetMsgServer is protocol independent, and NetMsgServers have been built to use various protocols. Of course, the NetMsgServers involved in a transfer must agree on the protocol used. Finally, the NetMsgServer on the destination computer uses that kernel's IPC to send the message to the correct destination task.

The ability to extend local IPC transparently across nodes is supported by the use of proxy ports. When a send right is transferred from one computer to another, the NetMsgServer on the destination computer creates a new port, or proxy, to represent the original port at the destination. Messages sent to this proxy are received by the NetMsgServer and are forwarded transparently to the original port. This procedure is one example of how NetMsgServers cooperate to make a proxy indistinguishable from the original port.

Because Mach is designed to function in a network of heterogeneous systems, it must provide a way for systems to send data formatted in a way that is understandable by both the sender and the receiver. Unfortunately, computers differ in the formats they use to store various types of data. For instance, an integer on one system might take 2 bytes to store, and the most significant byte might be stored before the least significant one. Another system might reverse this ordering. The NetMsgServer therefore uses the type information stored in a message to translate the data from the sender's to the receiver's format. In this way, all data are represented correctly when they reach their destination.

The NetMsgServer on a given computer accepts RPCs that add, look up, and remove network ports from the NetMsgServer's name service. As a security precaution, a port value provided in an add request for a port must match that in the remove request for a thread to ask for a port name to be removed from the database.

As an example of the NetMsgServer's operation, consider a thread on node A sending a message to a port that happens to be in a task on node B. The program simply sends a message to a port to which it has a send right. The message is first passed to the kernel, which delivers it to its first recipient, the NetMsgServer on node A. The NetMsgServer then contacts (through its database information) the NetMsgServer on node B and sends the message. The NetMsgServer on node B presents the message to the kernel with the appropriate local port for node B. The kernel finally provides the message to the receiving task when a thread in that task executes a `msg_receive()` call. This sequence of events is shown in Figure D.7.

Mach 3.0 provides an alternative to the NetMsgServer as part of its improved support for NORMA multiprocessors. The NORMA IPC subsystem of Mach 3.0 implements functionality similar to the NetMsgServer directly in the

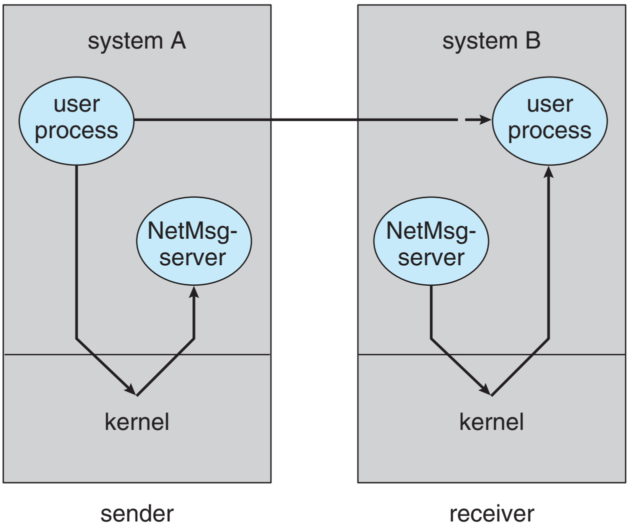


**Figure D.7** Network IPC forwarding by NetMsgServer.

Mach kernel, providing much more efficient internode IPC for multicomputers with fast interconnection hardware. For example, the time-consuming copying of messages between the NetMsgServer and the kernel is eliminated. Use of the NORMA IPC does not preclude use of the NetMsgServer; the NetMsgServer can still be used to provide Mach IPC service over networks that link a NORMA multiprocessor to other computers. In addition to the NORMA IPC, Mach 3.0 also provides support for memory management across a NORMA system and enables a task in such a system to create child tasks on nodes other than its own. These features support the implementation of a single-system-image operating system on a NORMA multiprocessor. The multiprocessor behaves like one large system rather than an assemblage of smaller systems (for both users and applications).

### D.5.4 Synchronization Through IPC

The IPC mechanism is extremely flexible and is used throughout Mach. For example, it may be used for thread synchronization. A port may be used as a synchronization variable and may have $n$ messages sent to it for $n$ resources. Any thread wishing to use a resource executes a receive call on that port. The thread will receive a message if the resource is available. Otherwise, it will wait on the port until a message is available there. To return a resource after use, the thread can send a message to the port. In this regard, receiving is equivalent to the semaphore operation `wait()`, and sending is equivalent to `signal()`. This method can be used for synchronizing semaphore operations among threads in the same task, but it cannot be used for synchronization among tasks, because only one task may have receive rights to a port. For more general-purpose semaphores, a simple daemon can be written to implement the same method.

## D.6 Memory Management

Given the object-oriented nature of Mach, it is not surprising that a principal abstraction in Mach is the memory object. Memory objects are used to manage secondary storage and generally represent files, pipes, or other data that are mapped into virtual memory for reading and writing (Figure D.8). Memory objects may be backed by user-level memory managers, which take the place of the more traditional kernel-incorporated virtual memory pager found in other operating systems. In contrast to the traditional approach of having the kernel manage secondary storage, Mach treats secondary-storage objects (usually files) as it does all other objects in the system. Each object has a port associated with it and may be manipulated by messages sent to its port. Memory objects —unlike the memory-management routines in monolithic, traditional kernels —allow easy experimentation with new memory-manipulation algorithms.

### D.6.1 Basic Structure

The virtual address space of a task is generally sparse, consisting of many holes of unallocated space. For instance, a memory-mapped file is placed in some set of addresses. Large messages are also transferred as shared memory segments. For each of these segments, a section of virtual memory address is used to provide the threads with access to the message. As new items are mapped or

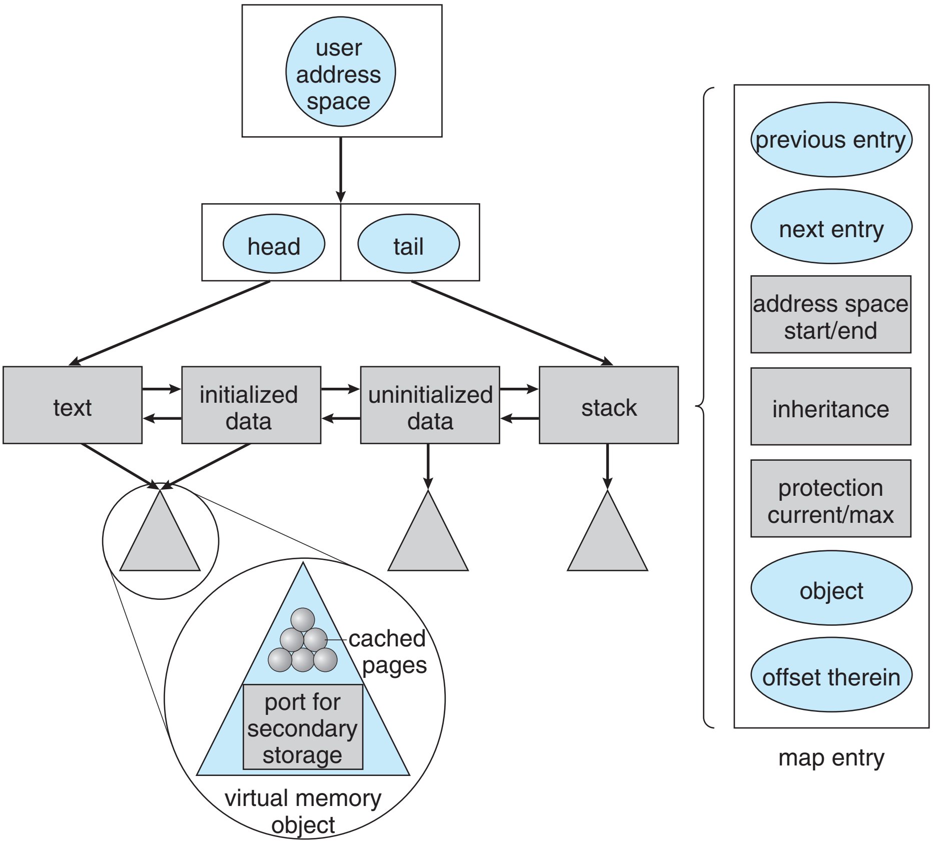

user address space
head
tail
text
initialized data
uninitialized data
stack
cached pages
port for secondary storage
virtual memory object
previous entry
next entry
address space start/end
inheritance
protection current/max
object
offset therein
map entry

**Figure D.8** Mach virtual memory task address map.

removed from the address space, holes of unallocated memory appear in the address space.

Mach makes no attempt to compress the address space, although a task may fail (or crash) if it has no room for a requested region in its address space. Given that address spaces are 4 GB or more, this limitation is not currently a problem. However, maintaining a regular page table for a 4-GB address space for each task, especially one with holes in it, would use excessive amounts of memory (1 MB or more). The key to sparse address spaces is that page-table space is used only for currently allocated regions. When a page fault occurs, the kernel must check to see whether the page is in a valid region, rather than simply indexing into the page table and checking the entry. Although the resulting lookup is more complex, the benefits of reduced memory-storage requirements and simpler address-space maintenance make the approach worthwhile.

Mach also has system calls to support standard virtual memory functionality, including the allocation, deallocation, and copying of virtual memory. When allocating a new virtual memory object, the thread may provide an address for the object or may let the kernel choose the address. Physical memory is not allocated until pages in this object are accessed. The object's backing store is managed by the default pager (Section D.6.2). Virtual memory objects are also allocated automatically when a task receives a message containing out-of-line data.

Associated system calls return information about a memory object in a task's address space, change the access protection of the object, and specify how an object is to be passed to child tasks at the time of their creation (shared, copy-on-write, or not present).

### D.6.2 User-Level Memory Managers

A secondary-storage object is usually mapped into the virtual address space of a task. Mach maintains a cache of memory-resident pages of all mapped objects, as in other virtual memory implementations. However, a page fault occurring when a thread accesses a nonresident page is executed as a message to the object's port. The concept that a memory object can be created and serviced by nonkernel tasks (unlike threads, for instance, which are created and maintained only by the kernel) is important. The end result is that, in the traditional sense, memory can be paged by user-written memory managers. When the object is destroyed, it is up to the memory manager to write back any changed pages to secondary storage. No assumptions are made by Mach about the content or importance of memory objects, so the memory objects are independent of the kernel.

In several circumstances, user-level memory managers are insufficient. For instance, a task allocating a new region of virtual memory might not have a memory manager assigned to that region, since it does not represent a secondary-storage object (but must be paged), or a memory manager might fail to perform pageout. Mach itself also needs a memory manager to take care of its memory needs. For these cases, Mach provides a default memory manager. The Mach 2.5 default memory manager uses the standard file system to store data that must be written to disk, rather than requiring a separate swap space, as in 4.3 BSD. In Mach 3.0 (and OSF/1), the default memory manager is capable of using either files in a standard file system or dedicated disk partitions. The default memory manager has an interface similar to that of the user-level ones, but with some extensions to support its role as the memory manager that can be relied on to perform pageout when user-level managers fail to do so.

Pageout policy is implemented by an internal kernel thread, the ***pageout daemon***. A paging algorithm based on FIFO with second chance (Section 10.4.5) is used to select pages for replacement. The selected pages are sent to the appropriate manager (either user level or default) for actual pageout. A user-level manager may be more intelligent than the default manager, and it may implement a different paging algorithm suitable to the object it is backing (that is, by selecting some other page and forcibly paging it out). If a user-level manager fails to reduce the resident set of pages when asked to do so by the kernel, the default memory manager is invoked, and it pages out the user-level manager to reduce the user-level manager's resident set size. Should the user-level manager recover from the problem that prevented it from performing its own pageouts, it will touch these pages (causing the kernel to page them in again) and can then page them out as it sees fit.

If a thread needs access to data in a memory object (for instance, a file), it invokes the `vm_map()` system call. Included in this system call is a port that identifies the object and the memory manager that is responsible for the region. The kernel executes calls on this port when data are to be read or written in that region. An added complexity is that the kernel makes these calls asynchronously, since it would not be reasonable for the kernel to be waiting on a user-level thread. Unlike the situation with pageout, the kernel has no recourse if its request is not satisfied by the external memory manager. The kernel has no knowledge of the contents of an object or of how that object must be manipulated.

Memory managers are responsible for the consistency of the contents of a memory object mapped by tasks on different machines. (Tasks on a single machine share a single copy of a mapped memory object.) Consider a situation in which tasks on two different machines attempt to modify the same page of an object at the same time. It is up to the manager to decide whether these modifications must be serialized. A conservative manager implementing strict memory consistency would force the modifications to be serialized by granting write access to only one kernel at a time. A more sophisticated manager could allow both accesses to proceed concurrently (for example, if the manager knew that the two tasks were modifying distinct areas within the page and that it could merge the modifications successfully at some future time). Most external memory managers written for Mach (for example, those implementing mapped files) do not implement logic for dealing with multiple kernels, due to the complexity of such logic.

When the first `vm_map()` call is made on a memory object, the kernel sends a message to the memory manager port passed in the call, invoking the `memory_manager_init()` routine, which the memory manager must provide as part of its support of a memory object. The two ports passed to the memory manager are a **control port** and a **name port**. The control port is used by the memory manager to provide data to the kernel—for example, pages to be made resident. Name ports are used throughout Mach. They do not receive messages but are used simply as points of reference and comparison. Finally, the memory object must respond to a `memory_manager_init()` call with a `memory_object_set_attributes()` call to indicate that it is ready to accept requests. When all tasks with send rights to a memory object relinquish those rights, the kernel deallocates the object's ports, thus freeing the memory manager and memory object for destruction.

Several kernel calls are needed to support external memory managers. The `vm_map()` call was just discussed. In addition, some commands get and set attributes and provide page-level locking when it is required (for instance, after a page fault has occurred but before the memory manager has returned the appropriate data). Another call is used by the memory manager to pass a page (or multiple pages, if read-ahead is being used) to the kernel in response to a page fault. This call is necessary since the kernel invokes the memory manager asynchronously. Finally, several calls allow the memory manager to report errors to the kernel.

The memory manager itself must provide support for several calls so that it can support an object. We have already discussed `memory_object_init()` and others. When a thread causes a page fault on a memory object's page, the kernel sends a `memory_object_data_request()` to the memory object's port on behalf of the faulting thread. The thread is placed in a wait state until the memory manager either returns the page in a `memory_object_data_provided()` call or returns an appropriate error to the kernel. Any of the pages that have been modified, or any “precious pages” that the kernel needs to remove from resident memory (due to page aging, for instance), are sent to the memory object via `memory_object_data_write()`. ***Precious pages*** are pages that may not have been modified but that cannot be discarded as they otherwise would be because the memory manager no longer retains a copy. The memory manager declares these pages to be precious and expects the kernel to return them when they are removed from memory. Precious pages save unnecessary duplication and copying of memory.

In the current version, Mach does not allow external memory managers to affect the page-replacement algorithm directly. Mach does not export the memory-access information that would be needed for an external task to select the least recently used page, for instance. Methods of providing such information are currently under investigation. An external memory manager is still useful for a variety of reasons, however:

- It may reject the kernel’s replacement victim if it knows of a better candidate (for instance, MRU page replacement).
- It may monitor the memory object it is backing and request pages to be paged out before the memory usage invokes Mach’s pageout daemon.
- It is especially important in maintaining consistency of secondary storage for threads on multiple processors, as we show in Section D.6.3.
- It can control the order of operations on secondary storage to enforce consistency constraints demanded by database management systems. For example, in transaction logging, transactions must be written to a log file on disk before they modify the database data.
- It can control mapped file access.

### D.6.3 Shared Memory

Mach uses shared memory to reduce the complexity of various system facilities, as well as to provide these features in an efficient manner. Shared memory generally provides extremely fast interprocess communication, reduces overhead in file management, and helps to support multiprocessing and database management. Mach does not use shared memory for all these traditional shared-memory roles, however. For instance, all threads in a task share that task’s memory, so no formal shared-memory facility is needed within a task. However, Mach must still provide traditional shared memory to support other operating-system constructs, such as the UNIX `fork()` system call.

It is obviously difficult for tasks on multiple machines to share memory and to maintain data consistency. Mach does not try to solve this problem directly; rather, it provides facilities to allow the problem to be solved. Mach supports consistent shared memory only when the memory is shared by tasks running on processors that share memory. A parent task is able to declare which regions of memory are to be ***inherited*** by its children and which are to be readable –writable. This scheme is different from copy-on-write inheritance, in which each task maintains its own copy of any changed pages. A writable object is addressed from each task’s address map, and all changes are made to the same copy. The threads within the tasks are responsible for coordinating changes to memory so that they do not interfere with one another (by writing to the same location concurrently). This coordination can be done through normal synchronization methods: critical sections or mutual-exclusion locks.

For the case of memory shared among separate machines, Mach allows the use of external memory managers. If a set of unrelated tasks wishes to share a section of memory, the tasks can use the same external memory manager and access the same secondary-storage areas through it. The implementor of this system would need to write the tasks and the external pager. This pager could be as simple or as complicated as needed. A simple implementation would allow no readers while a page was being written to. Any write attempt would cause the pager to invalidate the page in all tasks currently accessing it. The pager would then allow the write and would revalidate the readers with the new version of the page. The readers would simply wait on a page fault until the page again became available. Mach provides such a memory manager: the Network Memory Server (NetMemServer). For multicomputers, the NORMA configuration of Mach 3.0 provides similar support as a standard part of the kernel. This XMM subsystem allows multicomputer systems to use external memory managers that do not incorporate logic for dealing with multiple kernels. The XMM subsystem is responsible for maintaining data consistency among multiple kernels that share memory and makes these kernels appear to be a single kernel to the memory manager. The XMM subsystem also implements virtual copy logic for the mapped objects that it manages. This virtual copy logic includes both copy-on-reference among multicomputer kernels and sophisticated copy-on-write optimizations.

## D.7 Programmer Interface

A programmer can work at several levels within Mach. There is, of course, the system-call level, which, in Mach 2.5, is equivalent to the 4.3 BSD system-call interface. Version 2.5 includes most of 4.3 BSD as one thread in the kernel. A BSD system call traps to the kernel and is serviced by this thread on behalf of the caller, much as standard BSD would handle it. The emulation is not multithreaded, so it has limited efficiency.

Mach 3.0 has moved from the single-server model to support of multiple servers. It has therefore become a true microkernel without the full features normally found in a kernel. Rather, full functionality can be provided via emulation libraries, servers, or a combination of the two. In keeping with the definition of a microkernel, the emulation libraries and servers run outside the kernel at user level. In this way, multiple operating systems can run concurrently on one Mach 3.0 kernel.

An emulation library is a set of routines that lives in a read-only part of a program's address space. Any operating-system calls the program makes are translated into subroutine calls to the library. Single-user operating systems, such as MS-DOS and the Macintosh operating system, have been implemented solely as emulation libraries. For efficiency reasons, the emulation library lives in the address space of the program needing its functionality; in theory, however, it could be a separate task.

More complex operating systems are emulated through the use of libraries and one or more servers. System calls that cannot be implemented in the library are redirected to the appropriate server. Servers can be multithreaded for improved efficiency; BSD and OSF/1 are implemented as single multithreaded servers. Systems can be decomposed into multiple servers for greater modularity.

Functionally, a system call starts in a task and passes through the kernel before being redirected, if appropriate, to the library in the task's address space or to a server. Although this extra transfer of control decreases the efficiency of Mach, this decrease is balanced to some extent by the ability of multiple threads to execute BSD-like code concurrently.

At the next higher programming level is the C threads package. This package is a run-time library that provides a C language interface to the basic Mach threads primitives. It provides convenient access to these primitives, including routines for the forking and joining of threads, mutual exclusion through mutex variables (Section D.4.2), and synchronization through use of condition variables. Unfortunately, it is not appropriate for the C threads package to be used between systems that share no memory (NORMA systems), since it depends on shared memory to implement its constructs. There is currently no equivalent of C threads for NORMA systems. Other run-time libraries have been written for Mach, including threads support for other languages.

Although the use of primitives makes Mach flexible, it also makes many programming tasks repetitive. For instance, significant amounts of code are associated with sending and receiving messages in each task that uses messages (which, in Mach, is most tasks). The designers of Mach therefore provide an interface generator (or stub generator) called ***MIG***. MIG is essentially a compiler that takes as input a definition of the interface to be used (declarations of variables, types, and procedures) and generates the RPC interface code needed to send and receive the messages fitting this definition and to connect the messages to the sending and receiving threads.

## D.8 Summary

The Mach operating system is designed to incorporate the many recent innovations in operating-system research to produce a fully functional, technically advanced operating system.

The Mach operating system was designed with three critical goals in mind:

- Emulate 4.3 BSD UNIX so that the executable files from a UNIX system can run correctly under Mach.
- Have a modern operating system that supports many memory models and parallel and distributed computing.
- Design a kernel that is simpler and easier to modify than is 4.3 BSD.

As we have shown, Mach is well on its way to achieving these goals.

Mach 2.5 includes 4.3 BSD in its kernel, which provides the emulation needed but enlarges the kernel. This 4.3 BSD code has been rewritten to provide the same 4.3 functionality but to use the Mach primitives. This change allows the 4.3 BSD support code to run in user space on a Mach 3.0 system.

Mach uses lightweight processes, in the form of multiple threads of execution within one task (or address space), to support multiprocessing and parallel computation. Its extensive use of messages as the only communication method ensures that protection mechanisms are complete and efficient. By integrating messages with the virtual memory system, Mach also ensures that messages can be handled efficiently. Finally, by having the virtual memory system use messages to communicate with the daemons managing the backing store, Mach provides great flexibility in the design and implementation of these memory-object-managing tasks.

By providing low-level, or primitive, system calls from which more complex functions can be built, Mach reduces the size of the kernel while permitting operating-system emulation at the user level, much like IBM's virtual machine systems.

## Further Reading

The Accent operating system was described by [Rashid and Robertson (1981)]. A historical overview of the progression from an even earlier system, RIG, through Accent to Mach was given by [Rashid (1986)]. General discussions concerning the Mach model were offered by [Tevanian et al. (1989)].

[Accetta et al. (1986)] presented an overview of the original design of Mach. The Mach scheduler was described in detail by [Tevanian et al. (1987a)] and [Black (1990)]. An early version of the Mach shared memory and memory-mapping system was presented [Tevanian et al. (1987b)].

## Bibliography

**[Accetta et al. (1986)]** M. Accetta, R. Baron, W. Bolosky, D. B. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for UNIX Development", *Proceedings of the Summer USENIX Conference* (1986), pages 93–112.

**[Black (1990)]** D. L. Black, "Scheduling Support for Concurrency and Parallelism in the Mach Operating System", *IEEE Computer*, Volume 23, Number 5 (1990), pages 35–43.

**[Rashid (1986)]** R. F. Rashid, "From RIG to Accent to Mach: The Evolution of a Network Operating System", *Proceedings of the ACM/IEEE Computer Society, Fall Joint Computer Conference* (1986), pages 1128–1137.

**[Rashid and Robertson (1981)]** R. Rashid and G. Robertson, "Accent: A Communication-Oriented Network Operating System Kernel", *Proceedings of the ACM Symposium on Operating System Principles* (1981), pages 64–75.

**[Tevanian et al. (1987a)]** A. Tevanian, Jr., R. F. Rashid, D. B. Golub, D. L. Black, E. Cooper, and M. W. Young, "Mach Threads and the Unix Kernel: The Battle for Control", *Proceedings of the Summer USENIX Conference* (1987).

**[Tevanian et al. (1987b)]** A. Tevanian, Jr., R. F. Rashid, M. W. Young, D. B. Golub, M. R. Thompson, W. Bolosky, and R. Sanzi, "A UNIX Interface for Shared

Memory and Memory Mapped Files Under Mach", Technical report, Carnegie-Mellon University (1987).

**[Tevanian et al. (1989)]** A. Tevanian, Jr., and B. Smith, "Mach: The Model for Future Unix", *Byte* (1989).

# Credits

- Figure 1.14: From Hennesy and Patterson, *Computer Architecture: A Quantitative Approach, Third Edition*, ©2002, Morgan Kaufmann Publishers, Figure 5.3, p. 394. Reprinted with permission of the publisher.
- Figure 5.19: From Khanna/Sebree/Zolnowsky, "Realtime Scheduling in SunOS 5.0," Proceedings of Winter USENIX, January 1992, San Francisco, California. Derived with permission of the authors.
- Figure 5.30 adapted with permission from Sun Microsystems, Inc.
- Figure 10.20: From *IBM Systems Journal*, Vol. 10, No. 3, ©1971, International Business Machines Corporation. Reprinted by permission of IBM Corporation.
- Figure 12.5: Based on a table from *Pentium Processor User's Manual: Architecture and Programming Manual*, Volume 3, ©1993.
- Figure 14.8: From Leffler/McKusick/Karels/Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, ©1989 by Addison-Wesley Publishing Co., Inc., Reading, Massachusetts. Figure 7.6, p. 196. Reprinted with permission of the publisher.
- Figures 19.5, 19.6, and 19.8: From Halsall, *Data Communications, Computer Networks, and Open Systems, Third Edition*, ©1992, Addison-Wesley Publishing Co., Inc., Reading, Massachusetts. Figure 1.9, p. 14, Figure 1.10, p. 15, and Figure 1.11, p. 18. Reprinted with permission of the publisher.

# Index

**4-byte pages, 363, 364**
**32-byte memory, 363, 364**
**50-percent rule, 359**
**64-bit computing, 383**

A

**ABI (application binary interface), 78-79**
**aborting processes, 342**
**absolute code, 352**
**absolute path names, 546**
**abstract data type (ADT), 277-278**
**access, 539-541**
  anonymous, 605
  controlling, 552-554
  direct (relative), 539-541
  effective access time, 397-398
  kernel object, 884-885
  lightweight directory-access protocol, 607, 884
  memory, 15, 18, 19, 418-419, 498-500
  process migration for, 753
  and protection, 551
  random-access devices, 502
  random-access time, 450
  read, 292
  relative, 539-540
  Remote Access Tool, 625
  remote file, 764-767
  security access tokens, 662
  sequential, 539, 541
  wireless access points, 736
  write, 292
**access control:**
  discretionary, 684
  in Linux, 816-818
  MAC address, 745
  mandatory, 684-685
  role-based, 683-684
**access-control lists (ACLs), 552, 555, 826**
**accessed bits, 437**
**access mask, 849**
**access matrix, 675-685**
  defined, 675
  implementation of, 679-682
  and mandatory access control, 684-685
  and revocation of access rights, 682- 683
  and role-based access control, 683-684
**access rights, 534, 673, 680, 682-683**
**accounting, 110, 659, 788**
**ACG (Arbitrary Code Guard), 827**
**acknowledgment packet, 748**
**ACLs,** see **access-control lists**
**ACPI (advanced configuration and power interface), 516**
**activation record, 107**
**active directory, 607, 884**
**acyclic graphs, 547**
**acyclic-graph directories, 547-549**
**additional-reference-bits algorithm, 409-410**
**additional sense code, 512**
**additional sense-code qualifier, 512**
**address(es):**
  defined, 496
  linear, 380, 382
  logical, 353, 379
  MAC, 745
  physical, 354, 379
  trusted, 638
  virtual, 354
**address binding, 352-353**
**address mapping, 456-457**
**address resolution protocol (ARP), 745**

**address space:**
  logical vs. physical, 353-355
  virtual, 390, 391, 799-800
**address-space identifier (ASIDs), 366**
**address-space layout randomization (ASLR), 656, 827**
**Address Window Extension (AWE) memory, 894-895**
**admission-control algorithms, 230**
**ADT (abstract data type), 277-278**
**advanced configuration and power interface (ACPI), 516**
**advanced encryption standard (AES), 640**
**advanced local procedure call (ALPC), 138, 834**
**advanced technology attachment (ATA) buses, 456**
**advisory file-locking mechanisms, 535**
**AES (advanced encryption standard), 640**
**affinity, processor, 225-226**
**age, page, 800**
**aging, 213**
**ahead-of-time (AOT) compilation, 89, 90**
**alertable threads, 846**
**allocation:**
  buddy-system, 427, 428
  committing, 852
  contiguous, 356-360, 570-573
  equal, 414
  frame, 413-419
  free frames before and after, 364
  global, 415-418
  indexed, 575-577
  kernel memory, 426-430
  linked, 573-575
  local, 415-418
  over-, 401
  proportional, 414-415
  resource, 57
  of secondary storage, 570-578
  slab, 427-430, 797-798
**Allocation (data structure), 335, 336, 339**
**allocation problem, 358, 540, 571**
**ALPC (advanced local procedure call), 138, 834**
**altitudes, 863**
**AMD64 architecture, 382**
**Amdahl's Law, 164**
**AMD virtualization technology (AMD-V), 710-711**
**amplification write, 462**
**analytic evaluation, 245**
**Andrew file system (OpenAFS), 759**
**Android operating system, 89-91**
  process hierarchy, 122-123
  protection domain, 675
  RPC, 151-153
  thread pools, 178
  TrustZone, 670, 671
**anomaly detection, 656**
**anonymous access, 605**
**anonymous memory, 399, 469**
**anonymous pipes, 141-145**
**AOT (ahead-of-time) compilation, 89, 90**
**APCs (asynchronous procedure calls), 189-190, 846**
**APFS (Apple File System), 592**
**API (application program interface), 63-66.** See also specific types
**appending files 551**
**Apple File System (APFS), 592**
**application binary interface (ABI), 78-79**
**application component, 151-152**
**Application Container, 868**
**application containment, 703, 718-719**
**application frameworks layer (macOS and iOS), 87**
**application interface (I/O systems), 500-508**
  block and character devices, 503-504
  clocks and timers, 505-506
  network devices, 504-505
  nonblocking and asynchronous I/O, 506-507
**application layer (OSI model), 742**
**application programs (apps), 4, 75, 823**
  compatibility of, 830-831
  disinfection of, 658
  packaged, 859
  security of, 624
  specificity of, 77-79
  system services, 75
  user IDs for, 675
**application program interface (API), 63-66.** See also specific types
**application proxy firewalls, 660**
**application state, 378**
**Aqua interface, 59, 87**
**Arbitrary Code Guard (ACG), 827**
**architecture(s), 15-21**
  AMD64, 382
  ARMv8, 383-384, 671, 672
  big.LITTLE, 226-227
  clustered systems, 19-21
  IA-32, 379-382
  IA-64, 382
  multiprocessing, 124
  multiprocessor systems, 16-19

  NFS, 614
  single-processor systems, 15-16
  von Neumann, 12
  x86-64, 382
**Arduino, 70**
**argument vector, 787**
**armored viruses, 634**
**ARMv8 architecture, 383-384, 671, 672**
**ARP (address resolution protocol), 745**
**arrays:**
  redundant, see **RAID [redundant arrays of inexpensive disks]**
  storage, 472-473, 481
**ASICs, 46**
**ASIDs (address-space identifiers) 366**
**ASLR (address-space layout randomization), 656, 827**
**assignment edge, 323**
**asymmetric clustering, 19**
**asymmetric encryption, 641, 645**
**asymmetric encryption algorithm, 641**
**asymmetric multiprocessing, 220**
**asymmetry, in addressing, 129**
**asynchronous cancellation, 190**
**asynchronous devices, 502, 506-507**
**asynchronous message passing, 130**
**asynchronous procedure calls (APCs), 189-190, 846**
**asynchronous threading, 169**
**asynchronous writes, 585**
**ATA buses, 456**
**"at most once" functionality, 150**
**atomic instructions, 266, 269**
**atomic safe-save, 592**
**atomic variables, 269-270**
**attacks, 622**
  buffer-overflow, 628-631
  code-injection, 628-631
  code reuse, 827
  denial-of-service, 622, 636
  information leak, 827
  man-in-the-middle, 623, 635, 645
  replay, 622
  with tunneling, 659-660
  zero-day, 656
**attackers, 622**
**attack surface, 624**
**attributes, 551, 826, 875-876**
**attribute-definition table, 877**
**auditing, 659**
**audit trail, 669**
**augmented-reality applications, 42**
**authentication:**
  breaching of, 622
  and encryption, 641-644
  in Linux, 816
  multifactor, 653
  two-factor, 652
  user, 648-653
**automatic working-set trimming, 438**
**automount feature, 763**
**autoprobes, 785**
**availability, breach of, 622**
**Available (data structure), 334, 336, 338**
**AWE memory, 894-895**

B

**back door, 503, 626, 627, 638**
**background class, 186**
**background processes, 74-75, 115, 123, 215, 241**
**backing store, 376**
**back-pointers, 682**
**backups, 588-589**
**bad blocks, 466-467**
**bad-cluster file, 877**
**bad page, 856**
**balance, in multicore programming, 163**
**balanced binary search trees, 38**
**balloon memory manager, 721**
**bandwidth, 457**
**banker's algorithm, 333-337**
**barriers, memory, 265-266**
**based sections, 852**
**base file record, 876**
**base register, 351-352**
**bash (bourne-again shell), 58, 783**
**basic file systems, 564, 565**
**Bayes' theorem, 657**
**BCC (BPF Compiler Collection), 98-100**
**Belady's anomaly, 406**
**best-fit strategy, 358, 359**
**BGP (Border Gateway Protocol), 745**
**big cores, 226-227**
**big data, 22**
**big-endian, 150**
**big.LITTLE architecture, 226-227**
**binary format, 785**
**binary general tree, 38**
**binary search trees, 38, 39**
**binary semaphore, 273**
**binary translation, 708-710**
**binary trees, 38, 39**
**binders, 151**
**binding, 352**
**biometrics, 652-653**
**Bionic standard, 90**

**BIOS, 94**
**bit(s):**
  accessed, 437
  additional-reference-bits algorithm, 409-410
  contiguous, 432-433
  defined, 12
  mode, 24
  modify (dirty), 402
  reference, 409
  setuid, 674-675
  64-bit computing, 383
  valid-invalid, 368-369
**bit-level striping, 475**
**BitLocker, 863**
**bitmaps (bit vectors), 38-39, 579, 877**
**BKL, running on, 794**
**blade servers, 18-19**
**block(s), 186**
  bad, 466-467
  boot, 94, 464-466, 566
  boot control, 566
  defined, 564
  direct, 576
  disk, 40
  file-control, 565, 567
  index, 575-577
  indirect, 576, 577
  logical, 456
  process control, 109-110
  thread building, 186-188
  thread environment, 889-890
  TRIMing unused, 581-582
  virtual address control, 865
  volume control, 566
**block ciphers, 639**
**block devices, 502-504, 810-811**
**block device interface, 503**
**block groups, 806**
**blocking, indefinite 213**
**blocking I/O, 506**
**blocking (synchronous) message passing, 130**
**block-interleaved distributed parity, 477-478**
**block-level striping, 475**
**block number, relative, 540**
**block started by symbol (bss) field 108**
**block synchronization, 305**
**body (value), 187**
**boot block, 94, 465-466, 566**
**boot control block, 566**
**boot disk (system disk), 465**
**boot file 877**
**booting, 86, 94-95, 863-864, 872-874**
**boot loaders,** see **bootstrap programs**
**boot partition, 465**
**boot sector, 466**
**bootstrap port, 136**
**bootstrap programs (boot loaders, bootstrap loaders), 11, 70, 94, 465, 601**
**bootstrap server, 136**
**boot viruses, 632, 633**
**Border Gateway Protocol (BGP), 745**
**bottlenecks, 95**
**bottom half (interrupt service routines), 793-794**
**bounded buffer, 126**
**bounded-buffer problem, 290, 304**
**bounded capacity (of queue), 131-132**
**bounded waiting, 261**
**bourne-again shell (bash), 58, 783**
**BPF Compiler Collection (BCC), 98-100**
**breach of availability, 622**
**breach of confidentiality, 622**
**breach of integrity, 622**
**bridging, 723**
**broadcasting, 745**
**brokers, 837**
**browser process, 124**
**BSD UNIX, 49-50**
**bss (block started by symbol) field, 108**
**B+ tree (NTFS), 876**
**buddies, 427**
**buddy heap (Linux), 796**
**buddy system (Linux), 796**
**buddy-system allocation, 427, 428**
**buffers:**
  bounded and unbounded, 126
  bounded-buffer problem, 290, 304
  circular, 587, 716-717
  defined, 509
  translation look-aside, 365-368, 376, 384, 855
**buffer cache, 583-585**
**buffering, 131-132, 412, 499, 509-510**
**buffer-overflow attacks, 628-631**
**bugs, 66**
**bug bounty programs, 826**
**bus(es), 7, 456**
  advanced technology attachment, 456
  defined, 490-491
  eSATA, 456
  expansion, 490
  fibre channel, 456
  I/O, 456
  PCIe, 490

  - serial ATA, 456
  - serial-attached SCSI, 456, 490
  - universal serial, 456
- **busy waiting, 272, 493-494**
- **byte, 11**
- **bytecode, 727**
- **byte stream, 748**

## C

- **caches, 583-586**
  - buffer, 583-584
  - defined, 510
  - in Linux, 797, 798
  - location of, 765-766
  - as memory buffer, 350
  - page, 583, 798
  - and performance improvement, 583-586
  - policy for updating, 766-767
  - slabs in, 427, 428
  - unified buffer, 583-585
- **cache coherency, 32**
- **cache-consistency problem, 765**
- **cache management, 30-31**
- **cache manager (Windows 10), 864-866**
- **caching, 30-31, 510-511**
  - basic scheme, 764-765
  - client-side, 883
  - double, 584
  - write-back, 766
- **cancellation, thread, 190-192**
- **cancellation points, 191**
- **Canonical, 779**
- **capability(-ies), 680, 682-683, 685-686, 697**
- **capability-based protection systems, 685-687**
- **capability lists, 680-681**
- **capability systems, 826**
- **capacity, of queue, 131-132**
- **cascading termination, 121**
- **catching interrupts, 9, 494**
- **CAV (constant angular velocity), 457**
- **cd command, 751**
- **central processing units, 18, 318.** See also **entries beginning CPU**
- **Ceph, 484**
- **certificate authorities, 644**
- **CET (Control-flow Enforcement Technology), 828**
- **CFG (Control-Flow Guard), 827**
- **CFQ scheduler, 461, 811**
- **CFS,** see **Completely Fair Scheduler**
- **CFS (clustered file system), 768**
- **challenging passwords, 652**
- **change journal (Windows 10), 879**
- **character devices (Linux), 810-812**
- **character-stream devices, 502, 504**
- **character-stream interface, 504**
- **checksums, 462, 746**
- **children, 38, 111**
- **chip multithreading (CMT), 222, 223**
- **chipsets, 835**
- **Chrome, 124**
- **CIFS (common Internet file system), 607, 880**
- **ciphers, 639, 640**
- **circular buffers, 587, 716-717**
- **circularly linked lists, 37, 38**
- **circular SCAN (C-SCAN) scheduling algorithm, 460**
- **circular-wait condition (deadlocks), 321, 328-330**
- **claim edge, 333**
- **classes (Java), 694**
- **class loader, 727**
- **cleanup handler, 191**
- **clearing interrupts, 9, 494**
- **CLI (command-line interface), 56**
- **C library,** see **libc**
- **client(s), 73**
  - in client-server model, 606
  - defined, 757
  - diskless, 762
  - in distributed systems, 734
  - thin, 40
- **client-initiated approach to verifying cached data, 767**
- **client interface, 757**
- **client-server DFS model, 758-759**
- **client-server distributed system, 734**
- **client-server model, 606, 758-759, 861-862**
- **client-server systems, 42-43, 734**
- **client-side caching (CSC), 883**
- **client systems, 42**
- **clocks, 505-506**
- **clock algorithm, 410-411**
- **clock owner, 837**
- **clones, 591, 705**
- **clone() system call, 195-196**
- **closed-source operating systems, 46**
- **closures, 174, 186**
- **cloud computing, 44-45, 706**
- **cloud storage, 471, 751**
- **clusters, 19-21, 464, 574, 875**
- **cluster-based DFS model, 758-760**
- **clustered file system (CFS), 768**
- **clustered page tables, 374**
- **clustered systems, 19-21**

**clustering, 19, 20, 438**
**CLV (constant linear velocity), 456-457**
**CMT (chip multithreading), 222, 223**
**coarse-grained multithreading, 222**
**coaxial cables, 736**
**Cocoa framework, 87**
**Cocoa Touch, 87**
**code:**
  absolute, 352
  additional sense, 512
  byte-, 727
  error-correction, 462-463
  injection of, 628-631
  kernel, 261
  message-authentication, 643
  position-independent, 803
  reentrant (pure), 370
  relocatable, 353
**code-injection attack, 628-631**
**code integrity module (Windows 10), 828**
**code reuse attacks, 827**
**code review, 627, 628**
**code signing, 644, 690**
**codewords, 697**
**COM (Component Object Model), 882**
**com (top-level domain), 739**
**combined scheme index block, 576**
**command interpreter, 58-59**
**command-line interface (CLI), 56**
**committing allocations, 852**
**Common Criteria, 869**
**common Internet file system (CIFS), 607, 880**
**common name, 647**
**communication(s):**
  direct, 128
  indirect, 129
  inter-computer, 522
  interprocess, see **interprocess communication [IPC]**
  network, 738-749
    communication protocols, 741-745
    and naming/name resolution, 738-741
    TCP/IP example, 745-746
    UDP and TCP transport protocols, 746-749
  as operating system service, 57
  secure, with symmetric encryption, 639, 640
  systems programs for, 74
**communication links, 128**
**communication ports, 138**
**communication protocols, 741-745**
**communication system calls, 72-73**
**compaction, 360, 572**
**compare_and_swap() instruction, 267-269**
**compartmentalization, 669**
**compiler-based enforcement, 691-693**
**compile time, 352**
**Completely Fair Queuing (CFQ) scheduler, 461, 811**
**Completely Fair Scheduler (CFS), 236, 237, 790**
**complex messages, 136**
**Component Object Model (COM), 882**
**compression, 425-426, 757, 858, 878-879**
**compression ratio, 426**
**compression units, 878**
**computational kernels, 833**
**computation migration, 752**
**computation speedup, 123, 735, 753**
**computer programs,** see **application programs**
**computer system(s):**
  architecture of, 15-21
    clustered systems, 19-21
    multiprocessor systems, 16-19
    single-processor systems, 15-16
  distributed systems, 35-36
  firewalling to protect, 659-660
  I/O structure in, 14-15
  operating system viewed by, 5
  organization, 7-15
    interrupts, 8-11
    I/O structure, 14-15
    storage structure, 11-14
  process management in, 27-28
  protection in, 33-34
  real-time embedded systems, 45-46
  secure, 622
  security in, 33-34
  storage in, 11-14
  storage management in, 30, 32
  threats to, 634-637
**compute-servers system, 42-43**
**computing:**
  64-bit, 383
  cloud, 44-45, 706
  high-performance, 20
  mobile, 41-42
  peer-to-peer, 43-44
  safe, 658
  thin-client, 874-875
  traditional, 40-41
**computing environments, 40-46**
  client-server computing, 42-43
  cloud computing, 44-45
  mobile computing, 41-42

  peer-to-peer computing, 43-44
  real-time embedded systems, 45-46
  traditional, 40-41
  virtualization, 34-35
**concurrency, 163**
**Concurrency Runtime (ConcRT), 241-242, 890**
**concurrent dispatch queue, 185**
**conditional-wait construct, 281**
**condition variables, 278, 279, 302-303, 309-311, 889**
**confidentiality, breach of, 622**
**confinement problem, 678**
**conflict phase (of dispatch latency), 229**
**conflict-resolution mechanism (Linux), 784, 785**
**congestion control, 749**
**Connected Standby, 837**
**connectionless protocols, 747**
**connectionless (UDP) sockets, 147**
**connection-oriented protocols, 748**
**connection-oriented (TCP) sockets, 147**
**connection ports, 138**
**consistency, of distributed file systems, 767**
**consistency checker, 586-587**
**consistency checking, 586-587**
**consistency semantics, 608-609**
**consolidation, 706**
**constant angular velocity (CAV), 457**
**constant linear velocity (CLV), 456-457**
**consumer process, 126-127, 290, 291, 559-560**
**containers, 592, 718, 719**
**container objects (Windows 10), 664**
**containment, application, 703, 718-719**
**contaminants, 344**
**contended locks, 271**
**content-addressable storage, 484**
**contention scope, 217-218**
**context (of process), 114**
**context (of thread), 194**
**context switches, 114-115, 204**
**contiguous allocation, 356-360, 570-573**
**contiguous bit, 432-433**
**Control-flow Enforcement Technology (CET), 828**
**Control-Flow Guard (CFG), 827**
**controlled access, 552-554**
**controller(s), 456**
  defined, 491
  device, 456
  direct-memory-access, 498
  fibre channel bus, 491
  host, 456
**control partitions, 714**
**control programs, 5**
**control register, 492**
**convenience, 1**
**convoy effect, 207**
**cooperating processes, 123, 257**
**cooperative scheduling, 202**
**coordination, among processes, 260**
**copy-on-write technique, 399-401, 853**
**copy rights, access matrix with, 677**
**copy semantics, 510**
**core(s), 15-16, 18**
  big and little, 226-227
  dual-core design, 17, 18
  multicore processors, 221-224
  multicore programming, 162-166
  multicore systems, 16-18
  scheduling processes to run on, 199
**core dump, 95-96**
**core frameworks (macOS and iOS), 87**
**CoreUI, 825**
**counts, 533, 534**
**counters, 96-97**
  LRU page replacement with, 408
  program, 27, 106, 109
  timestamp, 845
**counting, 580**
**counting-based page replacement algorithm, 411-412**
**counting semaphore, 273**
**C program, memory layout in, 108**
**CPUs (central processing units), 18, 318**
**CPU-bound processes, 112**
**CPU burst, 201**
**CPU-I/O burst cycle, 201**
**CPU registers, 110**
**CPU scheduler, 113-114, 201**
**CPU scheduling, 24, 199-251**
  about, 201
  algorithms for, 205-217
    evaluation of, 244-249
    first-come, first-served scheduling of, 206-207
    multilevel feedback-queue scheduling of, 216-217
    multilevel queue scheduling of, 214-216
    priority scheduling of, 211-214
    round-robin scheduling of, 209-211
    shortest-job-first scheduling of, 207-209
  criteria, 204-205
  dispatcher, role of, 203-204

  - and I/O-CPU burst cycle, 201
  - multi-processor scheduling, 220-227
    - approaches to, 220-221
    - heterogeneous multiprocessing, 226-227
    - and load balancing, 224-225
    - and multicore processors, 221-224
    - and processor affinity, 225-226
  - operating-system examples, 234-244
    - Linux scheduling, 234-239
    - Solaris scheduling, 242-244
    - Windows scheduling, 239-242
  - preemptive vs. nonpreemptive scheduling, 202-203
  - real-time, 227-234
    - earliest-deadline-first scheduling, 232-233
    - and minimizing latency, 227-229
    - POSIX real-time scheduling, 233-234
    - priority-based scheduling, 229-230
    - proportional share scheduling, 233
    - rate-monotonic scheduling, 230-232
  - thread scheduling, 217-219
  - virtual machines, 720
- **CPU-scheduling information (PCBs), 110**
- **CPU utilization, 204**
- **crashes, 96**
- **crash dumps, 96**
- **CRCs (cyclic redundancy checks), 462**
- **creation:**
  - of files, 532, 542
  - of processes, 116-121
- **credentials, 787**
- **critical sections, 260**
- **critical-section object, 297, 888**
- **critical-section problem, 260-270**
  - Peterson's solution to, 262-265
  - and semaphores, 272-276
  - and synchronization hardware, 265-270
- **cryptography, 637-648**
  - defined, 638
  - and encryption, 638-645
    - asymmetric encryption, 641
    - authentication, 641-644
    - key distribution, 644-645
    - symmetric encryption, 639-640
  - implementation of, 645-646
  - TLS example of, 646-648
- **CSC (client-side caching), 883**
- **C-SCAN scheduling, 460**
- **C-SCAN scheduling algorithm, 460**
- **C shell, 58**
- **ctfs (file system), 598**
- **cumulative ACK, 748**
- **current directory, 546**
- **current-file-position pointer, 532**
- **cycle stealing, 499**
- **cyclic redundancy checks (CRCs), 462**
- **cylinder (hard disk drive), 450**
- **cylinder groups, 806**

D

- **d (page offset), 360**
- **DAC (discretionary access control), 684**
- **daemons, 22, 781**
- **daemon processes, 690**
- **daisy chain, 490**
- **DAM (Desktop Activity Moderator), 837**
- **dark web, 634**
- **Darwin operating system, 85, 88, 687-688**
- **data attributes, 875**
- **databases, 341, 842, 856**
- **data dependency, 164**
- **data-encryption standard (DES), 639**
- **Data Execution Prevention (DEP), 827**
- **datagrams, 743**
- **data-in register, 492**
- **data-link layer, 742**
- **data-link layer protocol, 645**
- **data loss, mean time of, 474**
- **data migration, 751-752**
- **data-out register, 492**
- **data parallelism, 165, 166**
- **data passing, between processes, 813**
- **data section (of process), 106**
- **data splitting, 164**
- **data striping, 475**
- **data view attribute, 862**
- **DCOM, 882**
- **DDoS attacks, 636**
- **deadline scheduler, 460, 461**
- **deadlock(s), 283-284, 317-343**
  - avoidance of, 326, 330-337
    - with banker's algorithm, 333-337
    - with resource-allocation-graph algorithm, 333
    - with safe-state algorithm, 331-333
  - characterization, 321-326
  - defined, 317
  - detection of, 337-341
  - methods for handling, 326-327
  - in multithreaded applications, 319-321
  - necessary conditions for, 321-323
  - prevention of, 326-330
  - recovery from, 341-343
  - system model for, 318-319

- system resource-allocation graphs for describing, 321-323
- **Debian, 779**
- **debuggers, 66**
- **debugging, 95-100, 165**
- **dedicated devices, 502**
- **deduplication, 757**
- **default access rights, 680**
- **default heap, 893**
- **default signal handlers, 189**
- **defense in depth, 653, 669**
- **deferred cancellation, 190**
- **deferred procedure calls (DPCs), 841, 847**
- **degree of multiprogramming, 112**
- **delayed revocation, 682**
- **delayed-write policy, 766**
- **deleting files, 532, 542, 551**
- **demand paging, 392-399, 430-436**
  - basic mechanism, 393-396
  - defined, 393
  - free-frame list, 396-397
  - with inverted page tables, 433
  - and I/O interlock, 434-436
  - and page size, 431-432
  - and performance, 397-399
  - and prepaging, 430-431
  - and program structure, 433-434
  - pure, 395
  - and TLB reach, 432-433
- **demand-zero memory, 799**
- **demilitarized zone (DMZ), 659**
- **denial-of-service (DOS) attacks, 622, 636**
- **dentry objects, 605, 804, 805**
- **DEP (Data Execution Prevention), 827**
- **DES (data-encryption standard), 639**
- **design of operating systems, 79-80**
  - distributed systems, 753-757
  - Linux, 780-783
  - Windows 10, 826-838
    - application compatibility, 830-831
    - dynamic device support, 837-838
    - energy efficiency, 836-837
    - extensibility, 833-834
    - international support, 835-836
    - performance, 831-833
    - portability, 838-839
    - reliability, 828-829
    - security, 826-828
- **desktop, 59**
- **Desktop Activity Moderator (DAM), 837**
- **Desktop Window Manager (DWM), 825**
- **detection-algorithm usage (deadlock), 340-341**
- **deterministic modeling, 245-247**
- **development kernels (Linux), 777**
- **device controllers, 456.** See also **I/O**
- **device directory,** see **directory(-ies)**
- **device drivers, 7, 490, 785**
- **Device Guard, 828**
- **device-management system calls, 71-72**
- **device objects, 863**
- **device reservation, 511**
- **device stacks, 862**
- **device-status table, 508-509**
- **DFSs,** see **distributed file systems**
- **digital certificates 644**
- **digital signatures, 643, 828**
- **digital-signature algorithm, 643**
- **dining-philosophers problem, 293-295**
- **dir command, 751**
- **direct access (files) 539-541**
- **direct blocks, 576**
- **direct communication, 128**
- **Direct-Compute, 825**
- **direct I/O, 504**
- **direct memory access (DMA), 15, 498-500**
- **direct-memory-access (DMA) controller, 498**
- **directory(-ies), 541-550**
  - active, 607, 884
  - acyclic-graph, 547-549
  - current, 546
  - fast sizing of, 592
  - file-system interface, 541-550
  - general graph, 549-550
  - implementation of, 568-570
  - lightweight directory-access protocol, 607
  - listing, 542
  - master file, 543
  - page, 381, 853
  - protecting, 554
  - root, 877
  - single-level, 542-543
  - tree-structured, 545-547
  - two-level, 543-545
  - user file, 543
- **directory object, 850**
- **direct virtual memory access (DVMA), 500**
- **dirty bits (modify bits), 402**
- **discretionary access control (DAC), 684**
- **disinfection, program, 658**
- **disk(s).** See also **mass-storage structure; RAID (redundant arrays of inexpensive disks)**
  - boot (system), 465
  - mini-, 704

  raw, 413, 464, 601
**disk arm, 450**
**disk blocks, 40**
**disk image, 723-724**
**diskless clients, 762**
**Disk Management tool, 465**
**disk-scheduling algorithms, 460-461**
**dispatched process, 112**
**dispatchers, 203-204, 239, 840-841**
**dispatcher database, 842**
**dispatcher objects, 297, 845-846**
**dispatching interrupts, 9, 494**
**dispatch latency, 203, 228, 229**
**dispatch queue, 185**
**distinguished name, 647**
**Distributed Denial-of-Service (DDoS) attacks, 636**
**distributed file systems (DFSs), 605, 757-768**
  client-server model, 758-759
  cluster-based model, 759-761
  defined, 757
  implementation techniques, 763-764
  naming in, 761-764
  remote file access in, 764-767
  trends in, 767-768
  Windows 10, 883
**distributed information systems (distributed naming services), 607**
**distributed lock manager (DLM), 21**
**distributed operating systems, 749-753**
**distributed systems, 35-36**
  advantages of, 733-735
  defined, 733
  design issues in, 753-757
  distributed file systems, 757-768
    client-server model, 758-759
    cluster-based model, 759-761
    defined, 757
    implementation techniques, 763-764
    naming in, 761-764
    remote file access in, 764-767
    trends in, 767-768
  distributed operating systems, 749-753
**distributions (GNU/Linux), 48**
**DLLs (dynamically linked libraries), 76, 355-356**
**DLM (distributed lock manager), 21**
**DMA (direct memory access), 15, 498-500**
**DMA-acknowledge, 499**
**DMA controller, 498**
**DMA-request, 499**
**DMZ (demilitarized zone), 659**
**DNS (domain-name system), 607, 739-740**
**dockers, 719**
**document(s):**
  File System Hierarchy Standard, 778-779
  living, 653
**domains:**
  capability lists for, 680-681
  protection, 671-675, 711
  public domain software, 779-780
  scheduling, 238
  security, 659
  Windows 10, 884
**domain-name system (DNS), 607, 739-740**
**domain switching, 673, 674**
**DOS attacks, 622, 636**
**double buffering, 499, 509**
**double caching, 584**
**double indirect blocks, 576**
**doubly linked lists, 37**
**down time, 572**
**DPCs (deferred procedure calls), 841, 847**
**DRAM (dynamic random-access memory), 11**
**drive formatting, 463, 464**
**drive mirroring, 476**
**driver end (STREAM), 519**
**driver objects, 863**
**driver-registration system (Linux), 784, 785**
**Drive Writes Per Day (DWPD), 453**
**dropped packets:**
  TCP transfer with, 748, 749
  UDP transfer with, 747-748
**dual-booted systems, 601**
**dual-core design, 17, 18**
**dual-mode operation, 24-25**
**DVMA (direct virtual memory access), 500**
**DWM (Desktop Window Manager), 825**
**DWPD (Drive Writes Per Day), 453**
**dynamically linked libraries (DLLs), 76, 355-356**
**dynamic device support (Windows 10), 837-838**
**dynamic linking, 803**
**dynamic loading, 355**
**dynamic protection, 673**
**dynamic random-access memory (DRAM), 11**
**dynamic storage-allocation problem, 358, 571**
**dynamic tick, 836-837**

E

**earliest-deadline-first (EDF) scheduling, 232-233**
**ease of use, 4, 822**
**easily remembered passwords, 651**
**eBPF tracing tool, 99, 100**
**ec2, 44**
**ECC (error-correction code), 462-463**
**economic benefits of multithreaded processes, 162**
**EDF (earliest-deadline-first scheduling, 232-233**
**edu (top-level domain), 739**
**effective access time, 397-398**
**effective capabilities, 685**
**effective memory-access time, 367**
**effective transfer rates, 451, 486**
**effective UID, 34**
**efficiency, 1, 582-583, 692, 836-837**
**electrical storage systems, 14**
**elevator algorithm,** see **SCAN scheduling algorithm**
**ELF (Executable and Linkable Format), 76-77, 801, 802**
**embedded computers, 5**
**empty processes, 123**
**empty slabs, 429, 798**
**emulation, 34, 717-718**
**emulators, 703**
**encapsulation (Java), 696**
**encrypted viruses, 633**
**encryption, 638-645**
  asymmetric, 641, 645
  authentication, 641-644
  defined, 638
  key distribution, 644-645
  public-key, 641
  symmetric, 639-640
**energy efficiency, 836-837**
**enhanced second-chance page-replacement algorithm, 410-411**
**entitlements (Darwin), 686-687**
**entry section, 260**
**entry set, 303, 304, 307**
**environment:**
  computing, 40-46
  kernel, 88
  operating system as, 4
  programming, 703, 717
  run-time, 64-65
  thread environment blocks, 889-890
**environment vector, 787**
**equal allocation, 414**
**errors, 462-463, 467, 511-512**
**error-correcting organization, 476-477**
**error-correction code (ECC), 462-463**
**error detection, 57, 462**
**error handling, 511-512**
**eSATA buses, 456**
**escalate privileges, 34**
**escape (operating systems), 503**
**Ethernet packets, 745-746**
**events, 297**
**event latency, 227-228**
**event objects (Windows 10), 845**
**event tracing, 822**
**event-vector table, 11**
**eVM, 729**
**"exactly once" functionality, 150**
**exceptions, 22, 497, 847-848**
**exception dispatcher, 847**
**exclusive locks, 534**
**exec() system call, 188, 786-789**
**Executable and Linkable Format,** see **ELF**
**executable files 75, 107, 530**
**executing files, 551**
**execution of user programs, 801-803**
**execution time, 353**
**executive (Windows 10), 848-874**
  booting, 872-874
  cache manager, 864-866
  facilities for client-server computing, 861-862
  I/O manager, 862-864
  object manager, 849-851
  plug-and-play manager, 869-870
  power manager, 870-871
  process manager, 858-860
  registry, 871-872
  security reference monitor, 866-869
  virtual memory manager, 851-858
**exit section, 260**
**exit() system call, 121-122**
**expansion bus, 490**
**exponential average, 208**
**export list, 612**
**ext2 (second extended file system), 805**
**ext3 (third extended file system), 805-807**
**ext4 (fourth extended file system), 805**
**extended file attributes, 531**
**extended file system (extfs), 566, 805**
**extensibility, of Windows 10, 833-834**
**extent (contiguous space), 572**
**external data representation (XDR), 150**
**external fragmentation, 359-360, 571-572**
**extfs (extended file system), 566, 805**

**F**

**failure(s), 473, 474, 754-756**
**failure analysis, 95-96**
**failure modes (remote file systems), 607-608**
**fairness parameter, 307**
**fair scheduling, 791**
**false negatives, 656**
**false positives, 656**
**fast directory sizing, 592**
**fast I/O mechanism, 865-866**
**fast-user switching, 825, 874-875**
**FAT (file-allocation table), 574-575**
**fault, page, 394-395**
**fault tolerance, 19, 754**
**fault-tolerant systems, 754**
**FC (fiber channel), 470**
**FCB (file-control block), 565, 567**
**FC buses, 456**
**FC bus controller, 491**
**FCFS scheduling, 458, 459**
**FCFS scheduling algorithm, 206-207, 458**
**fd (file descriptor), 568, 788**
**fences, memory, 266**
**fibers, 241, 889-890**
**fiber optic cables, 736**
**fibre channel (FC), 470**
**fibre channel (FC) buses, 456**
**fibre channel (FC) bus controller, 491**
**fidelity, 704**
**FIFO, 38**
**FIFO page replacement algorithm, 404-406**
**50-percent rule, 359**
**file(s), 29-30, 529-530.** See also **directory(-ies);** specific types
  appending, 551
  attributes of, 530-531
  defined, 255, 527, 529
  deleting, 532, 542, 551
  executing, 551
  internal structure of, 537-539
  locking, 534-536
  opening, 532
  operations on, 532-536
  paging, 851
  reading, 532, 551
  renaming, 542
  searches for, 541, 542
  truncating, 532
  writing, 532, 551
**file-allocation table (FAT), 574-575**
**file-control block (FCB), 565, 567**
**file descriptor (fd), 568, 788**
**file handle, 568**
**file info window (macOS), 531**
**file management, 74**
**file-management system calls, 71**
**file mapping, 555, 557**
**file migration, 761-762**
**file modification, 74**
**file objects, 605, 804-805, 862**
**file-open count, 534**
**file-organization module, 565**
**file pointers, 534**
**file reference, 876**
**file replication, 761**
**file-server system, 43**
**file sessions, 608**
**file sharing, 602-603**
**file systems, 597-616**
  Andrew, 759
  Apple, 592
  basic, 564, 565
  clustered, 768
  common Internet, 607, 880
  consistency semantics in, 608-609
  defined, 564
  distributed, see **distributed file systems (DFSs)**
  extended, 566, 805
  file sharing, 602-603
  Google, 759-761
  Hadoop, 484
  Linux, 803-810
    ext3 file system, 805-807
    journaling, 808
    /proc file system, 808-810
    virtual, 804-805
  log-based transaction-oriented, 587-588
  logical, 565
  mounting of, 598-602
  network, 610-615
  network file, 610-615, 759
  operations, 566-568
  parallel, 768
  partitions in, 601-602
  registration of, 785
  remote, 605-608
  Solaris, 482-484, 597, 599
  special-purpose, 597-598
  structure, 564-566
  traversing, 542
  traversing of, 542
  UNIX, 565-566, 598
  usage, 568
  virtual, 603-605, 804-805
  Windows 10, 875-879
  write-anywhere file layout, 589-593

  ZFS, 482-484, 581, 588, 598
**file-system context, 788**
**File System Hierarchy Standard document, 778-779**
**file system implementation, 563-593**
  allocation methods, 570-578
    contiguous allocation, 570-573
    indexed allocation, 575-577
    linked allocation, 573-575
    performance, 577-578
  directory implementation, 568-570
  efficiency, 582-583
  file-system operations, 566-568
  file-system structure, 564-566
  free-space management, 578-582
  performance, 583-586
  recovery, 586-589
  WAFL example, 589-593
**file-system interface, 529-560**
  access methods, 539-541, 551-554
  directory structure, 541-550
    acyclic-graph directories, 547-549
    general graph directory, 549-550
    single-level directory, 542-543
    tree-structured directories, 545-547
    two-level directory, 543-545
  file attributes, 530-531
  and file concept, 529-530
  file operations, 532-536
  file structure, 537-539
  file types, 536-537
  memory-mapped files, 555-560
  protection, 550-555
**file-system management, 29-30**
**file-system manipulation (operating system service), 56-57**
**file table, 788**
**file transfer, 750-751**
**file viruses, 632**
**filter drivers, 863**
**filtering system-call, 688**
**filter management, 863**
**fine-grained multithreading, 222**
**Finish (data structure), 335, 339**
**firewalls, 41, 659-660**
**firewall chains, 815**
**firewall management, 815**
**firmware, 11, 12**
**first-come first-served (FCFS) scheduling algorithm, 206-207, 458**
**first-fit strategy, 358, 359**
**first-level interrupt handler (FLIH), 496**
**first readers, 291**
**flash translation layer (FTL), 453-454**
**flexibility, of compiler-based enforcement, 692**
**FLIH (first-level interrupt handler), 496**
**flow control, 519, 748, 749**
**flushing 366**
**folder redirection, 883**
**folders, 59**
**foreground priority separation boost, 843**
**foreground processes, 115, 122, 215, 241**
**fork() and exec() process model (Linux), 786-789**
**fork-join model, 180-184**
**fork()system call, 118-119, 188, 786-789**
**formatting, 463, 464**
**forwarding, 466**
**forward-mapped page tables, 371**
**4-byte pages, 363, 364**
**four-layered model of security, 623-625**
**fourth extended file system (ext4), 805**
**fragments, packet, 815**
**fragmentation, 359-360, 571-572**
**frame(s), 360**
  in data-link layer, 742
  free, 364, 396-397, 425-426
  minimum number of, 413-414
  page, 853
  page faults vs., 404, 405
  victim, 402
**frame allocation, 413-419**
  allocation algorithms, 403, 414-415
  equal, 414
  global vs. local, 415-418
  minimum number of frames, 413-414
  non-uniform memory access, 418-419
  proportional, 414-415
**frame-allocation algorithm, 403, 414-415**
**frame table, 365**
**free-behind technique, 585**
**FreeBSD, 70, 71**
**free frames, allocation and, 364**
**free-frame list, 396-397, 425-426**
**free page, 856**
**Free Software Foundation (FSF), 48**
**free-space list, 578-579**
**free-space management, 578-582**
**fresh value, 647**
**front-end processors, 522**
**FSF (Free Software Foundation), 48**
**fsgid property, 818**
**fsuid property, 818**
**FTL (flash translation layer), 453-454**
**full backup, 589**
**full slabs, 429, 798**

functional programming languages, 313-314

G

Galois field math, 478
Gantt chart, 206
garbage collection, 454, 549, 550, 727
GB (gigabyte), 11
gcc (GNU C compiler), 778
GCD (Grand Central Dispatch), 185-186
GDT (global descriptor table), 379
general graph directories, 549-550
general revocation, 682
gestures, 60
get command, 751
GFS (Google file system), 759-761
gigabyte (GB), 11
git, 50
global allocation, 415-418
global descriptor table (GDT), 379
global dispatch queues, 185
global replacement, 415-418
global table, 679
GNOME desktop, 60
GNU C compiler (gcc), 778
GNU General Public License (GPL), 48
GNU/Linux, 48
Google Android, 42
Google file system (GFS), 759-761
GPFS, 768
GPL (GNU General Public License), 48
GPU (graphics processing unit), 735
graceful degradation, 19
Grand Central Dispatch (GCD), 185-186
granularity, minimum, 791
graphs, acyclic, 547
graphical user interfaces (GUIs), 56, 59-61
graphics processing unit (GPU), 735
green threads, 167
group (user class), 551
group identifiers 33-34
grouping, 580
group policies, 884
group rights (Linux), 817
GRUB, 94
guard pages, 852
guest (operating system), 34
guest processes, 702
GUIs (graphical user interfaces), 56, 59-61

H

hackers, 622
Hadoop, 22
Hadoop distributed file system (HDFS), 759, 760
Hadoop file system (HDFS), 484
HAL (hardware-abstraction layer), 835, 840
handles, 849
handle tables, 849
handling, signal, 188-189
handshake, three-way, 748
hard affinity, 225
hard-coding techniques, 129
hard disk drives (HDDs), 13
  components of, 450-451
  defined, 449
  scheduling, 457-461
hard errors, 463, 467
hard limits, 438, 857
hard links, 532, 549, 879
hard page faults, 416
hard real-time systems, 227
hardware, 4
  instructions, 266-269
  I/O system, 490-500
    direct memory access, 498-500
    interrupts, 494-498
    polling, 493-494
  and main memory, 350-352
  and memory management, 28
  process migration and, 753
  for relocation and limit registers, 357
  for storing page tables, 365-368
  synchronization, 265-270
  transformation of requests to operations by, 516-519
  virtual machine, 710-713
hardware-abstraction layer (HAL), 835, 840
hardware objects, 672
hardware threads, 222
hardware transactional memory (HTM), 312
hard working-set limits, 438
hash collision, 39
hashed page tables, 373-374
hash functions, 38-39, 643
hash map, 39
hash tables, 570
hash value (message digest), 643
HBA (host bus adapter), 491
HDDs, see hard disk drives
HDFS (Hadoop distributed file system), 759, 760
HDFS (Hadoop file system), 484

**head crash, 451**
**heaps, 893-894**
**heap section (of process), 107**
**heartbeat procedure, 754-755**
**heterogeneous multiprocessing, 226-227**
**hibernation, 870**
**hierarchical paging, 371-373**
**high-availability service, 19**
**high contention, 271, 286**
**high memory, 795**
**high-performance computing, 20**
**high-performance event timer (HPET), 505**
**high priority, 212**
**hijacking, session, 623**
**hit ratio, 367, 432**
**hives, 871**
**hold-and-wait condition (deadlocks), 321, 327-328**
**holes, 358**
**HoloLens, 874**
**honeypot, 655-656**
**horizontal scalability, 484**
**host(s):**
  distributed system, 734
  operating system, 35, 177
  virtual machine, 702
**host-attached storage, 470**
**host bus adapter (HBA), 491**
**host controller, 456**
**host-id, 739**
**host name, 73, 738**
**hot spare (drive), 480**
**hot-standby mode, 19, 20**
**HPET (high-performance event timer), 505**
**HTM (hardware transactional memory), 312**
**HTTP protocol, 881**
**huge pages, 363, 432**
**Hybrid Boot, 873-874**
**hybrid cloud, 44**
**hybrid operating systems, 86-91**
**hypercalls, 717**
**hypercall interface, 839**
**hyper-threading, 222, 832**
**Hyper-V for Client, 831**
**hyper-V hypervisor, 839**
**hypervisors, 670, 702**
  separation, 729
  type 0, 702, 713-714
  type 1, 703, 714-715
  type 2, 703, 715-716

I

**IA-32 architecture, 379-382**
**IA-64 architecture, 382**
**IaaS (infrastructure as a service), 44**
**IB (InfiniBand) 473**
**icons, 59**
**ideal processors, 242, 842**
**idempotent, 759**
**identifiers**
  address-space, 366
  file, 530
  group, 33-34
  host names vs., 738
  location-independent file, 764
  process, 116
  spoofed, 606
  user, 33
**idle process, 872**
**idle threads, 239, 842**
**IKE protocol, 646**
**image, disk, 723-724**
**immediate revocation, 682**
**immutable shared files, 609**
**immutable-shared-file semantics, 609**
**imperative languages, 313**
**impersonation, 867**
**implementation:**
  CPU scheduling algorithm, 249
  cryptography, 645-646
  directory, 568-570
  file system, see **file system implementation**
  monitor, 280-281
  of naming techniques, 763-764
  of operating systems, 80-81
  Pthread, 169, 170
  of security defenses, 653-662
    and accounting, 659
    and auditing, 659
    and firewalling, 659-660
    and intrusion prevention, 655-657
    levels of defenses, 661-662
    and logging, 659
    and security policy, 653
    and virus protection, 657-659
    and vulnerability assessment, 653-655
  semaphore, 274-276
  synchronization primitive, 845-846
  virtual machine, 713-719
    application containment, 718-719
    emulation, 717-718
    paravirtualization, 716-717

    - programming-environment virtualization, 717
    - type 0 hypervisors, 713-714
    - type 1 hypervisors, 714-715
    - type 2 hypervisors, 715-716
    - and virtual machine life cycle, 713
- **implicit threading, 176-188**
  - fork join, 180-183
  - Grand Central Dispatch, 185-186
  - Intel thread building blocks, 186-188
  - OpenMP and, 183-185
  - thread pools and, 177-180
- **importance hierarchy (Android), 122**
- **include file 40**
- **increase scheduling priority privilege, 887**
- **incremental backup, 589**
- **indefinite blocking (starvation), 213, 343**
- **independence, location, 762, 769**
- **independent processes, 123**
- **indexes, 540, 542, 576**
- **index blocks, 575-577**
- **indexed allocation, 575-577**
- **index root, 876**
- **indirect blocks, 576, 577**
- **indirect communication, 129**
- **indirection, 683, 703**
- **InfiniBand (IB), 473**
- **information leak attacks, 827**
- **information-maintenance system calls, 72**
- **information sharing, 123**
- **infrastructure as a service (IaaS), 44**
- **inheritable capabilities, 685**
- **init process, 117**
- **in-memory file-system structures, 568, 569**
- **inode, 482, 565, 577**
- **inode objects, 605, 804**
- **input/output,** see **I/O**
- **input/output operations per second (IOPS), 461**
- **insert() method, 305-307**
- **InServ storage array, 481**
- **instruction register, 12**
- **integrity, 622, 687-688**
- **integrity label (Windows 10), 663**
- **integrity levels, 826**
- **Intel processors, 379-382**
  - event-vector table, 496
  - IA-32 architecture, 379-382
  - IA-64 architecture, 382
  - thread building blocks, 186-188
- **inter-computer communications, 522**
- **interface(s).** See also specific types
  - choice of, 60-62
  - defined, 501
  - speeds of, 510
- **interlock, I/O, 434-436**
- **intermachine interface, 757**
- **internal fragmentation, 359**
- **international support (Windows 10), 835-836**
- **Internet, 737**
- **Internet Key Exchange (IKE), 646**
- **Internet model, 742**
- **Internet Protocol (IP), 743-746.** See also **Transmission Control Protocol/Internet Protocol (TCP/IP)**
- **Internet Service Providers (ISPs), 737**
- **interpreted languages, 717**
- **interprocess communication (IPC), 123-153**
  - in client-server systems, 145-153
    - remote procedure calls, 149-153
    - sockets, 146-149
  - in Linux, 777, 812-813
  - Mach example of, 135-138
  - in message-passing systems, 127-132
  - pipes in, 139-145
  - POSIX shared-memory example of, 132-135
  - in shared-memory systems, 125-127
  - Windows example of, 138-139
- **interrupt(s), 8-11, 494-498**
  - defined, 494
  - in Linux, 794
  - maskable, 10, 495-496
  - nonmaskable, 10, 495
  - software (traps), 497
  - in Windows 10, 846-848
- **interrupt chaining, 10, 496**
- **interrupt-controller hardware, 9, 495**
- **interrupt-dispatch table (Windows 10), 848**
- **interrupt-handler routine, 9, 494-495**
- **interrupt latency, 228**
- **interrupt objects, 848**
- **interrupt priority levels, 10-11, 497**
- **interrupt request levels (IRQLs), 841, 846**
- **interrupt-request line, 9, 494**
- **interrupt service routines (ISRs), 844**
- **interrupt vector, 9, 496**
- **intruders, 622**
- **intrusion prevention, 655-657**
- **intrusion-prevention systems (IPSs), 656**
- **inverted page tables, 374-375, 433**
- **involuntary context switches, 204**
- **I/O (input/output):**
  - fast mechanism for, 865-866

  raw, 464, 465
  structure of, 14-15
  in virtual machines, 722-723
**I/O-bound processes, 112**
**I/O burst, 201**
**I/O bus, 456**
**I/O channel, 522**
**I/O control level (file system), 564**
**I/O interlock, 434-436**
**I/O manager, 862-864**
**I/O operations, 56**
**IOPS (input/output operations per second), 461**
**I/O request packet (IRP), 863**
**iOS operating system, 42, 87-89**
**I/O status information (PCBs), 110**
**I/O subsystem(s), 32-33**
  kernels in, 508-516
  procedures supervised by, 516
**I/O system(s), 489-525**
  application interface, 500-508
    block and character devices, 503-504
    clocks and timers, 505-506
    network devices, 504-505
    nonblocking and asynchronous I/O, 506-507
    vectored I/O, 507-508
  hardware, 490-500
    direct memory access, 498-500
    interrupts, 494-498
    for memory-mapped I/O, 491-493
    polling, 493-494
    summary, 500
  kernel subsystem, 508-516
    buffering, 509-510
    caching, 510-511
    data structures, 512-514
    error handling, 511-512
    I/O scheduling, 508-509
    power management, 514-516
    procedures supervised by, 516
    protection, 512
    spooling and device reservation, 511
  Linux, 810-812
  overview, 489-490
  STREAMS mechanism, 519-521
  and system performance, 521-524
  transformation of requests to hardware operations, 516-519
**IP (Internet Protocol), 743-746.** See also **Transmission Control Protocol/Internet Protocol (TCP/IP)**
**IPC,** see **interprocess communication**
**iPhone, 60**
**IPIs (Windows 10), 847-848**
**IPSec, 646**
**IPSs (intrusion-prevention systems), 656**
**IRP (I/O request packet), 863**
**IRQLs (interrupt request levels), 841, 846**
**iSCSI, 471**
**ISPs (Internet Service Providers), 737**
**ISRs (interrupt service routines), 844**
**Itanium, 382**
**iteration space, 187**

## J

**Java:**
  DNS lookup in, 740
  file locking in, 534, 535
  fork-join in, 180-184
  Lambda expressions in, 174
  language-based protection in, 694-696
  synchronization, 303-311
    condition variables, 309-311
    monitors, 303-307
    reentrant locks, 307-308
    semaphores, 308-309
  thread dumps in, 339
  thread pools in, 179-180
**Java Executor interface, 175-176**
**Java threads, 173-176**
**Java Virtual Machine (JVM), 177, 717, 727-728**
**JBOD (Just a Bunch of Disks), 472**
**JIT compilers, 728**
**jobs, processes vs., 106**
**job objects, 859**
**job scheduling, 106**
**journaling, 587-588, 808**
**Just a Bunch of Disks (JBOD), 472**
**just-in-time (JIT) compilers, 728**
**JVM,** see **Java Virtual Machine**

## K

**KB (kilobyte), 11**
**K Desktop Environment (KDE), 60**
**Kerberos network authentication protocol, 607**
**kernel(s), 6, 7, 501, 508-516**
  buffering, 509-510
  caching, 510-511
  computational, 833
  data structures, 36-40, 512-514
  error handling, 511-512
  I/O scheduling, 508-509
  and I/O subsystems, 516

  - Linux, 776-778, 781
  - nonpreemptive, 262
  - power management, 514-516
  - preemptive, 262
  - protection, 512
  - secure, 839-840
  - spooling and device reservation, 511
  - synchronization of, 295-299, 792-794
  - uni-, 728
  - Windows 10, 839-848

**kernel abstractions, 89**
**kernel code, 261**
**kernel data structures, 36-40, 512-514**
**kernel environment, 88**
**kernel extensions (kexts), 89**
**kernel memory allocation, 426-430**
**kernel mode, 24, 25, 782**
**Kernel-Mode Driver Framework (KMDF), 864**
**kernel-mode threads (KT), 841**
**kernel modules, 86, 783-786**
**kernel module management, 784**
**kernel object access (Windows 10), 884-885**
**kernel threads, 166, 217, 234**
**kernel virtual memory, 801**
**Kernighan's Law, 98**
**kexts (kernel extensions), 89**
**keys:**
  - for capabilities, 683
  - defined, 638
  - Internet Key Exchange, 646
  - in lock-key schemes, 681
  - master, 683
  - private, 641
  - public, 641
  - sense, 512
  - session, 647

**key distribution, 644-645**
**key ring, 644**
**keystreams, 640**
**keystroke logger, 634**
**kilobyte (KB), 11**
**KMDF (Kernel-Mode Driver Framework), 864**
**Korn shell, 58**
**KT (kernel-mode threads), 841**
**Kubernetes, 719**

L

**labels, for mandatory access control, 685**
**Lambda expressions, 174**
**languages, 313-314, 717**
**language-based protection systems, 690-696**
  - compiler-based enforcement, 691-693
  - in Java, 694-696

**LANs (local-area networks), 36, 735-737**
**large objects, 430**
**latency:**
  - dispatch, 203, 228, 229
  - event, 227-228
  - interrupt, 228
  - in real-time systems, 227-229
  - rotational, 451
  - target, 236, 791

**latency command, 494**
**layers (of network protocols), 645**
**layered approach (operating system structure), 83-84**
**layered protocols, 891**
**LBA (logical block address), 456**
**LCNs (logical cluster numbers), 875**
**LDAP (lightweight directory-access protocol), 607, 884**
**LDT (local descriptor table), 379**
**least-frequently used (LFU) page-replacement algorithm, 411-412**
**least privilege, principle of, 626, 627, 668-669**
**least-recently-used (LRU) algorithm, 407-408**
**left child, 38**
**LFH design, 894**
**LFU page-replacement algorithm, 411-412**
**lgroups, 419**
**libc (C library), 63, 69, 370, 781**
**libraries:**
  - C, 63, 69, 370, 781
  - Linux system, 781
  - shared, 356, 392
  - thread, 168-176
    - about, 168-169
    - Java, 173-176
    - Pthreads, 169-171
    - Windows, 171-173

**library operating systems, 728**
**licensing, Linux, 779-780**
**life cycle:**
  - I/O request, 518-519
  - virtual machine, 713

**lifetime, virtual address space, 799-800**
**LIFO, 37-38**
**lightweight directory-access protocol (LDAP), 607, 884**
**lightweight process (LWP), 193**

**limit register, 351-352**
**linear addresses, 380, 382**
**linear lists (files) 569-570**
**line discipline, 811-812**
**link(s):**
  communication, 128
  defined, 548
  hard, 532, 549, 879
  resolving, 548
  symbolic, 879
**linked allocation, 573-575**
**linked lists, 37, 38, 579-580**
**linked scheme index block, 576**
**linkers, 75, 76**
**linking, 355-356, 803, 882**
**Linux, 48, 775-819**
  capabilities in, 685-686
  design principles for, 780-783
  file systems, 803-810
    ext3 file system, 805-807
    journaling, 808
    /proc file system, 808-810
    virtual, 804-805
  history of, 775-780
  input and output, 810-812
  interprocess communication, 812-813
  kernel modules, 783-786
  lockdep tool, 330
  memory management, 795-803
    execution and loading of user programs, 801-803
    physical memory, 795-798
    virtual memory, 436-437, 798-801
  network structure, 813-815
  process management, 786-790
  process representation in, 111
  scheduling in, 234-239, 790-794
  security model, 816-818
  swap-space management in, 468-470
  synchronization in, 296-298
  system structure, 83
  threads example, 195-196
  tree of processes, 116
  Windows subsystem for, 91
**Linux distributions, 776, 779**
**Linux instance, 91**
**Linux kernel, 776-778, 781**
**Linux kernel data structures, 40**
**Linux system(s), 776**
  components of, 781-783
  history of, 778-779
  obtaining page size on, 364
**Linux timers, 27**
**lists, 37**
  access, 679-680
  access-control, 552, 555, 826
  capability, 680-681
  export, 612
  free-frame, 396-397, 425-426
  free-space, 578-579
  linear, 569-570
  linked, 37, 38, 579-580
  user control, 561
**listing directories, 542**
**listing file names and attributes, 551**
**little cores, 227**
**little-endian, 150**
**Little's formula, 247**
**live CD, 48**
**live DVD, 48**
**livelock, 320-322**
**live migration (virtual machines), 706, 724-726**
**liveness, 283-284**
**living documents, 653**
**loadable kernel modules (LKMs), 86**
**load balancing, 224-225, 735, 753**
**loaders, 75-77, 695, 727, 783.** See also **bootstrap programs**
**loading, 355, 801-803**
**load sharing, 220**
**load time, 353**
**local allocation, 415-418**
**local-area networks (LANs), 36, 735-737**
**local descriptor table (LDT), 379**
**locality model, 421**
**locality of reference, 395**
**locality property, 857**
**local-name, 763**
**local replacement, 415-418**
**local replacement algorithm, 420-421**
**location, file 530, 534**
**location independence, 761, 762**
**location-independent fil identifiers 764**
**location transparency, 761**
**locks, 681.** See also **deadlock(s)**
  advisory, 535
  exclusive, 534
  in Java API, 534, 535
  mandatory, 535
  mutex, 270-272, 299-300
  nonrecursive, 299
  Pushlocks, 831
  reader-writer, 292-293
  reentrant, 307-308
  scope of, 305
  shared, 534
  for shared data, 70

- **lock-free algorithms, 284**
- **locking, page, 434-436**
- **locking files, 534-536**
- **lock-key scheme, 681**
- **lofs (file system), 598**
- **log-based transaction-oriented file systems, 587-588**
- **log files, 95, 876**
- **log-file service, 878**
- **logging, 57, 659**
- **logging area, 878**
- **logical address, 353, 379**
- **logical address space, 353-355**
- **logical blocks, 456**
- **logical block address (LBA), 456**
- **logical cluster numbers (LCNs), 875**
- **logical file system, 565**
- **logical formatting, 464**
- **logical memory, 24, 362.** See also **virtual memory**
- **logical processors, 832**
- **logical records, 539**
- **logic bomb, 627**
- **login, remote, 750**
- **loopback, 148**
- **loosely coupled system, 83**
- **loosely-coupled systems, 19**
- **love bug virus, 658**
- **low contention, 271**
- **low-fragmentation heap (LFH) design, 894**
- **low-level formatting (disks), 463**
- **low priority, 212**
- **LRU algorithm, 407-408**
- **LRU-approximation page replacement algorithm, 409-411**
- **LRU page replacement, 407-409**
- **ls command, 751**
- **Lustre, 768**
- **LWP (lightweight process), 193**
- **LXC containers, 718, 719**

## M

- **MAC (mandatory access control), 684-685**
- **MAC (message-authentication code), 643**
- **MAC address, 745**
- **Mach-O format, 77**
- **Mach operating system, 84, 135-138**
- **macOS operating system:**
  - GUI, 61
  - as hybrid system, 87-89
  - latency command, 494
  - sandboxing in, 690-691
- **macro viruses, 632**
- **magic number (files) 537**
- **magnetic tapes, 455**
- **mailboxes, 129-130**
- **main memory, 349-385**
  - and address binding, 352-353
  - ARMv8 architecture, 383-384
  - contiguous allocation of, 356-360
  - and dynamic linking, 355-356
  - and dynamic loading, 355
  - and hardware, 350-352
  - Intel 32 and 64-bit architectures, 379-382
  - and logical vs. physical address space, 353-355
  - paging for management of, 360-376
    - basic method, 360-365
    - hardware, 365-368
    - hashed page tables, 373-374
    - hierarchical paging, 371-373
    - inverted page tables, 374-375
    - and Oracle SPARC Solaris, 375-376
    - protection, 368-369
    - and shared pages, 369-371
    - swapping with, 377
  - shared libraries, 356
  - and swapping, 376-378
- **main queue, 185**
- **main TLB, 384**
- **major page faults, 416**
- **malware, 625-628**
- **MANs (metropolitan-area networks), 36**
- **mandatory access control (MAC), 684-685**
- **mandatory file-locking mechanisms, 535**
- **mandatory policy, 826**
- **man-in-the-middle attack, 623, 635, 645**
- **many-to-many multithreading model, 167-168**
- **many-to-one multithreading model, 166-167**
- **mapping, 39**
  - address, 456-457
  - file, 555, 557
  - memory, 802-803
- **MapReduce system, 22, 761**
- **marshaling, 150, 882**
- **Mars Pathfinder, 285**
- **maskable interrupts, 10, 495-496**
- **masquerading, 622, 623, 635**
- **mass-storage management, 30**
- **mass-storage structure, 449-486**
  - address mapping, 456-457
  - attachment of storage, 469-473
  - device management, 463-467
  - error detection and correction, 462-463

  - hard disk drives, 450-451, 457-461
  - nonvolatile memory devices, 452-454, 461-462
  - overview, 449-450
  - RAID, 473-485
    - extensions, 481-482
    - for object storage, 483-485
    - performance improvement, 475
    - problems with, 482-483
    - RAID levels, 475-481
    - reliability improvement, 473-475
  - scheduling, 457-462
  - secondary storage connection methods, 456
  - swap-space management, 467-469
  - volatile memory, 454-455

**master book record (MBR), 465, 466**

**master file directory (MFD), 543**

**master file table, 566**

**master key, 683**

**master secret (TLS), 647**

**matchmakers, 151**

**Max (data structure), 335**

**maximum number of resources, declaration of, 330**

**MB (megabyte), 11**

**MBR (master book record), 465, 466**

**MD5 message digest, 643**

**mean time between failures (MTBF), 473, 474**

**mean time of data loss, 474**

**mean time to repair, 474**

**mechanical storage systems, 14**

**mechanisms, 80, 668.** See also specific mechanisms

**medium access control (MAC) address, 745**

**medium objects, 430**

**megabyte (MB), 11**

**memory:**
  - Address Window Extension, 894-895
  - anonymous, 399, 469
  - defined, 14
  - demand-zero, 799
  - direct memory access, 14, 498-500
  - direct virtual memory access, 500
  - high, 795
  - in-memory file-system structures, 568,569
  - layout of, in C program, 108
  - layout of process in, 106
  - logical, 24, 362
  - main, see **main memory**
  - network virtual memory, 765
  - over-allocation of, 401
  - physical, 24, 362, 390, 391, 795-798
  - secondary, 395
  - semiconductor, 14
  - shared, 57, 73, 123, 125, 556-560
  - software transactional, 312
  - 32-byte, 363, 364
  - transactional, 311-312
  - virtual, see **virtual memory**
  - volatile, 454-455

**memory access:**
  - direct, 15, 498-500
  - direct virtual, 500
  - effective memory-access time, 367
  - non-uniform, 18, 19, 418-419

**memory-address register, 354**

**memory allocation, 358-359, 426-430**

**memory barriers, 265-266**

**memory compression, 425-426, 858**

**memory devices:**
  - management of, 463-467
  - nonvolatile, 452-454
    - defined, 449
    - NAND flash controller algorithms for, 453-454
    - overview, 452-453
    - scheduling, 461-462

**memory fences, 266**

**memory management, 28-29**
  - in Linux, 795-803
    - execution and loading of user programs, 801-803
    - physical memory, 795-798
    - virtual memory, 798-801
  - with virtual machines, 721-722
  - in Windows 10, 892-895

**memory-management information (PCBs), 110**

**memory-management unit (MMU), 354, 855**

**memory manager (MM), 851**

**memory-mapped files, 555-560, 892-893**

**memory-mapped I/O, 491-493**

**memory mapping (Linux), 802-803**

**memory model, 265-266**

**memory protection, 357, 368-369**

**memory stall, 221-222**

**memory-style error-correcting organization, 476-477**

**memory transactions, 311**

**messages, 135**
  - complex, 136
  - in distributed systems, 734
  - OSI network, 742, 744

  in Win32 API, 891
**message-authentication code (MAC), 643**
**message digest (hash value), 643**
**message modification 622-623**
**message passing, 123, 125, 130**
**message-passing model, 57, 72-73, 127-132**
  buffering, 131-132
  Mach example, 135-138
  naming, 128-130
  synchronization, 130-131
**metadata, 607, 876**
**metaslabs, 581**
**methods (Java), 694**
**Metro, 823**
**metropolitan-area networks (MANs), 36**
**MFD (master file directory), 543**
**MFU page-replacement algorithm, 412**
**microkernels, 84-86**
**Microsoft Interface Definition Language (MIDL), 150, 882**
**Microsoft Windows,** see **Windows operating system (generally)**
**micro TLBs, 384**
**middleware, 6, 7**
**MIDL (Microsoft Interface Definition Language), 150, 882**
**migration:**
  computation, 752
  data, 751-752
  file, 761-762
  process, 752-753
  push and pull, 224
  with virtual machines, 706, 724-726
**minidisks, 704**
**minifilters 863**
**minimum granularity, 791**
**miniport driver, 864**
**minor page faults, 416**
**mirrored volume, 474**
**mirroring, 474, 476**
**MM (memory manager), 851**
**MMU (memory-management unit), 354, 855**
**mobile computing, 41-42**
**mobile systems, 115, 377-378**
**mode bits, 24**
**moderate contention, 285**
**Modern, 823**
**modified page, 856**
**modify bits (dirty bits), 402**
**modularity, 123**
**modules:**
  file-organization, 565
  kernel, 86, 783-786
  pluggable authentication, 816
  stream, 519
**module loader, 783**
**module-management system, 783, 784**
**module unloader, 783**
**monitors, 276-282**
  dining-philosophers solution with, 295,296
  implementation of, using semaphores, 280-281
  in Java, 303-307
  resumption of processes within, 281-282
  security reference, 866-869
  usage of, 277-280
**monitor calls,** see **system calls**
**monitor type, 277**
**monoculture, 634**
**monolithic operating systems, 82-83**
**Moore's Law, 5**
**most-frequently used (MFU) page-replacement algorithm, 412**
**motherboard, 20**
**motivation, for multithreading, 160-161**
**mounting, 464, 598-602**
**mount points, 598, 879**
**mount protocol (NFS), 612**
**mount table, 517, 567**
**MTBF (mean time between failures), 473, 474**
**MUI support, 840**
**multicore processors, 221-224**
**multicore programming, 162-166**
**multicore systems, 16-18**
**multidimensional RAID level 6, 478**
**multifactor authentication, 653**
**multilevel feedback-queue scheduling algorithm, 216-217**
**multilevel index, 576**
**multilevel queue scheduling algorithm, 214-216**
**multimode operation, 25-26**
**multipartite viruses, 634**
**Multiple UNC Provider (MUP), 882-883**
**multiple user interface (MUI) support, 840**
**multiprocessing, 16, 220, 226-227, 794**
**multiprocessors, 18, 220**
**multi-processor scheduling, 220-227**
  approaches to, 220-221
  examples, 234-242
    Linux, 234-239
    Solaris, 242-244
    Windows, 239-242
  heterogeneous multiprocessing, 226-227

- and load balancing, 224-225
- and multicore processors, 221-224
- and processor affinity, 225-226

**multiprocessor systems, 16-19**
**multiprogramming, 23, 112, 420**
**multi-provider router, 883**
**multitasking, 23, 115, 790**
**multithreaded processes, 160**
- benefits of, 162
- deadlocks in, 319-321
- and exec() system call, 188
- and fork() system call, 188
- models of, 166-168
- motivation for, 160-161
- and signal handling, 188-189

**multithreading:**
- chip, 222, 223
- coarse-grained, 222
- fine-grained, 222
- many-to-many, 167-168
- many-to-one, 166-167
- one-to-one, 167
- simultaneous, 222

**multi-touch hardware, 874**
**MUP (Multiple UNC Provider), 882-883**
**mutex locks, 270-272, 299-300**
**mutual exclusion (mutex), 260, 267, 268, 845**
**mutual-exclusion condition (deadlocks), 321, 327**

## N

**names:**
- common and distinguished, 647
- host, 73, 738
- resolution of, 738-741

**named condition variables, 309**
**named pipes, 143-145, 881-882**
**named semaphores, 300-301**
**named shared-memory object, 559**
**name server, 739**
**namespaces, 787**
**naming, 128-130**
- defined, 761
- in distributed file systems, 761-764
- distributed naming services, 607
- domain name system, 607
- file, 530
- and network communication, 738-741
- port, 135-136

**naming schemes, 763**
**naming structures (DFS), 761-763**
**NAND flas controller algorithms, 453-454**
**NAS (network-attached storage), 470-471**
**NAT (network address translation), 723**
**national-language-support (NLS) API, 835**
**NDIS (Network Device Interface specification) 880**
**Need (data structure), 335, 336**
**need-to-know principle, 672**
**nested page tables (NPTs), 710, 712**
**network(s):**
- communication structure, 738-749
  - communication protocols, 741-745
  - and naming/name resolution, 738-741
  - TCP/IP example, 745-746
  - UDP and TCP transport protocols, 746-749
- defined, 36
- firewalling to protect, 659-660
- in Linux, 813-815
- local-area, 36, 735-737
- metropolitan-area, 36
- network operating systems, 749-751
- personal-area, 36
- as resource types, 318
- security in, 623
- storage-area, 21, 470, 472
- structure of, 735-738
- threats to, 634-637
- virtual private, 646, 881
- wide-area, 36, 735, 737-738
- wireless, 41, 736-737

**network address translation (NAT), 723**
**network-attached storage (NAS), 470-471**
**network computers, 40**
**network devices, 504-505, 810**
**Network Device Interface specificatio (NDIS), 880**
**network file system (NFS), 610-615, 759**
**network information service (NIS), 607**
**networking, 880-884**
**network interfaces (Windows 10), 880**
**network layer, 742**
**network-layer protocol, 645**
**network operating systems, 36, 749-751**
**network protocols, registration of, 785**
**network time protocol (NTP), 505**
**network virtual memory, 765**
**new state, 108**
**NFS (network file system), 610-615, 759**
**NFS protocol, 612-614**
**nice value, 236, 790**
**NIS (network information service), 607**

**NLS API, 835**
**no-access page, 852**
**nonblocking I/O, 506-507**
**nonblocking message passing, 130**
**noncontainer objects (Windows 10), 664**
**nonmaskable interrupts, 10, 495**
**nonpreemptive kernels, 262**
**nonpreemptive scheduling, 202**
**nonrecursive locks, 299**
**nonrepudiation, 644**
**nonresident attributes, 876**
**nonsignaled state, 297**
**non-uniform memory access (NUMA), 18, 19, 418-419**
**nonvolatile memory (NVM) devices, 13, 14, 452-454**
  defined, 449
  NAND flash controller algorithms for, 453-454
  overview, 452-453
  scheduling, 461-462
**nonvolatile storage (NVS), 14, 449**
**NOOP scheduler, 461**
**no-preemption condition (deadlocks), 321, 328**
**Normal World, 838**
**notify() method, 305-307**
**Notify port, 135**
**NPTs (nested page tables), 710, 712**
**NTFS, 875-877**
**NTP (network time protocol), 505**
**NUMA,** see **non-uniform memory access**
**NUMA-aware algorithms, 225-226**
**NUMA mode, 238-239**
**NVM devices,** see **nonvolatile memory devices**
**NVM express (NVMe), 456**
**NVS (nonvolatile storage), 14, 449**

## O

**objects:**
  access lists for, 679-680
  in cache, 428
  container, 664
  critical-section, 297, 888
  defined, 672
  dentry, 605, 804, 805
  device, 863
  directory, 850
  dispatcher, 297, 845-846
  driver, 863
  event, 845
  file, 605, 804-805, 862
  hardware, 672
  inode, 605, 804
  interrupt, 848
  job, 859
  in Linux, 797, 804
  named shared-memory, 559
  noncontainer, 664
  section, 139, 852
  semaphore, 845
  sharing, 885-886
  small, medium, and large, 430
  software, 672
  superblock, 605, 804, 805
  timer, 845
  in Windows 10, 664, 845-846, 848, 849
**Object Linking and Embedding (OLE), 882**
**object manager (Windows 10), 849-851**
**object storage, 483-485**
**object types, 849, 850**
**objfs (file system), 598**
**off-line compaction of space, 572**
**OLE (Object Linking and Embedding), 882**
**one-time passwords, 652**
**one-to-one multithreading model, 167**
**on-line compaction of space, 572**
**OOM (out-of-memory) killers, 418**
**OpenAFS (Andrew file system), 759**
**open count, 533**
**open-file table, 533, 567**
**opening files, 532**
**OpenMP, 183-185, 312-313**
**open operating systems, 634**
**open-source operating systems, 46-51**
**Open Systems Interconnection (OSI) model, 741-744**
**operating system(s):**
  application specificity to, 77-79
  booting, 94-95
  building, 92-93
  closed-source, 46
  computing environments, 40-46
  CPU scheduling in, 234-244
    Linux scheduling, 234-239
    Solaris scheduling, 242-244
    Windows scheduling, 239-242
  debugging, 95-100
  defined, 1, 3, 5-7
  design goals for, 79-80
  features of, 3
  functioning of, 4-7
  implementation of, 80-81
  kernel data structures, 36-40

  linkers and loaders, 75-77
  network, 36
  open-source, 46-51
  operations, 21-27
    dual-mode and multimode, 24-26
    multiprogramming and multitasking, 23-24
    and timer, 26-27
  reasons for studying, 6
  as resource allocator, 5
  resource management by, 27-32
  security in, 623-624
  services provided by, 55-58
  structure, 81-91
    hybrid systems, 86-91
    layered approach, 83-84
    microkernels, 84-86
    modules, 86
    monolithic, 82-83
  study of, 50
  system calls, 62-74
    and API, 63-66
    functioning of, 62-63
    types of calls, 66-74
  system services, 74-75
  system's view of, 5
  threads in, 194-196
  user interface with, 4-5, 58-62
  virtualization components, 719-726
    CPU scheduling, 720
    I/O devices, 722-723
    live migration, 724-726
    memory management, 721-722

**optimal page replacement, 406-407**

**optimal page replacement algorithm, 406-407**

**optimistic approach, 285**

**Oracle SPARC Solaris, 375-376**

**Orange Book, 826**

**ordinary pipes, 140-143**

**org (top-level domain), 739**

**orphan processes, 122**

**OS/2 operating system, 821-822**

**OSI model, 741-744**

**OSI network model, 741-744**

**OSI protocol stack, 742-744**

**other users (class), 551**

**out-of-band key delivery, 644**

**out-of-memory (OOM) killers, 418**

**over-allocation of memory, 401**

**overcommitment, 720**

**over-provisioning, 454**

**owners:**
  clock, 837
  file, 603
  as user class, 551

**owner rights, 678, 817**

P

**p (page number), 360**

**PaaS (platform as a service), 44**

**packaged applications, 859**

**package systems, 823**

**PAE (page address extension), 381**

**pages.** See also specific types
  defined, 360
  locking, 434-436
  obtaining size of, 364

**page address extension (PAE), 381**

**page allocator (Linux), 796**

**page-buffering algorithms, 412**

**page cache, 583, 798**

**page directory, 381, 853**

**page-directory entries (PDEs), 853**

**page directory pointer table, 381**

**page faults, 394-395, 405, 416**

**page-fault-frequency (PFF), 424-425**

**page-fault rate, 398, 423**

**page frames, 853**

**page-frame number (PFN) database, 856**

**page in, 377**

**page locking, 434-436**

**page number (p), 360**

**page offset (d), 360**

**page out, 377**

**pageout policy (Linux), 800**

**pageout process (Solaris), 439**

**page replacement, 401-413.** See also **frame allocation**
  and application performance, 412-413
  basic mechanism, 401-404
  counting-based page replacement, 411-412
  defined, 401
  FIFO page replacement, 404-406
  global vs. local, 415-418
  LRU-approximation page replacement, 409-411
  LRU page replacement, 407-409
  optimal page replacement, 406-407
  and page-buffering algorithms, 412

**page replacement algorithm, 403**

**page size, 363, 364, 431-432**

**page slots, 469**

**page table(s), 361-378, 393**
  clustered, 374
  defined, 361

- for demand paging, 395
- forward-mapped, 371
- hardware for storing, 365-368
- hashed, 373-374
- for hierarchical paging, 371-373
- inverted, 374-375, 433
- nested, 710, 712
- Oracle SPARC Solaris, 375-376

**page-table base register (PTBR), 365**
**page-table entries (PTEs), 853**
**page-table length register (PTLR), 369**
**paging, 360-376**
- basic method, 360-365
- demand, 392-399, 430-436
  - basic mechanism, 393-396
  - defined, 393
  - free-frame list, 396-397
  - with inverted page tables, 433
  - and I/O interlock, 434-436
  - and page size, 431-432
  - and performance, 397-399
  - and prepaging, 430-431
  - and program structure, 433-434
  - pure, 395
  - and TLB reach, 432-433
- hardware, 365-368
- for hashed page tables, 373-374
- hierarchical, 371-373
- IA-32, 380-381
- inverted, 374-375
- in Linux, 800
- and memory protection, 368-369
- and Oracle SPARC Solaris, 375-376
- priority, 440
- and shared pages, 369-371
- swapping with, 377, 378

**paging files 851**
**paired passwords, 652**
**PAM (pluggable authentication modules), 816**
**PAN (personal-area network), 36**
**parallel file system (PFS), 768**
**parallelism, 163, 165-166, 475**
**parallelization, 20**
**parallel regions, 183-184**
**paravirtualization, 703, 716-717**
**parent-child relationship, 140**
**parent process, 111**
**partial revocation, 682**
**partial slabs, 429, 798**
**partitions:**
- boot, 465
- control, 714
- file-system, 601-602
- raw, 468
- root, 601
- storage device, 463-465
- variable-partition schemes, 358

**partition boot sector, 566**
**partitioning, device, 463-464**
**passphrases, 651**
**passwords, 554, 649-652**
**path names, 544, 546**
**path-name translation (NFS), 614-615**
**PB (petabyte), 12**
**PCBs (process control blocks), 109-110**
**PCIe bus, 490**
**PC motherboard, 20**
**PCS (process-contention scope), 217-218**
**PC systems, 874**
**PDEs (page-directory entries), 853**
**peer-to-peer computing, 43-44**
**peer-to-peer distributed systems, 734**
**PE (Portable Executable) format, 77**
**penetration test, 654**
**performance:**
- and allocation of secondary storage, 578-579
- and demand paging, 397-399
- and file system implementation, 583-586
- and I/O system, 521-524
- and page replacement, 412-413
- RAID structure to improve, 475
- under swapping, 378
- virtualization requirement related to, 704
- of Windows 10, 831-833

**performance monitoring, 96-97**
**performance tuning, 95-97**
**periodic processes, 230**
**periodic task rate, 230**
**permanent revocation, 682**
**permissions, 553, 669**
**permitted capabilities, 685**
**per-process open-file table, 567**
**per-process tools, 96, 97**
**personal-area network (PAN), 36**
**personal computer (PC) systems, 874**
**personal firewalls, 660**
**personal identification number (PIN), 652**
**personalities, 87, 787**
**pessimistic approach, 285**
**petabyte (PB), 12**
**Peterson's solution, 262-265**
**PFF (page-fault-frequency), 424-425**
**PFN database, 856**
**PFS (parallel file system), 768**
**phishing, 624**

**PHY (ports), 490**
**physical address, 354, 379**
**physical address space, 353-355**
**physical formatting, 463**
**physical layer, 741, 742**
**physical memory, 24, 362, 390, 391, 795-798**
**physical security, 623**
**physical-to-virtual (P-to-V) conversion, 724**
**PIC (position-independent code), 803**
**Pico process, 91**
**Pico Providers, 823**
**pid (process identifier) 116, 787**
**PIN (personal identification number), 652**
**pinning, 436, 866**
**PIO (programmed I/O), 498**
**pipes, 139-145**
  anonymous, 141-145
  implementation considerations, 139-140
  named, 143-145, 881-882
  ordinary, 140-143
  use of, 146
**pipe mechanism, 813**
**platform as a service (PaaS), 44**
**platter (disks), 450**
**PLM (Process Lifetime Manager), 837**
**plug-and-play and (PnP) managers, 869-870**
**pluggable authentication modules (PAM), 816**
**plug-in process, 124**
**PnP managers, 869-870**
**PoFX (power framework), 870**
**Point-to-Point Tunneling Protocol (PPTP), 881**
**policy(-ies), 80**
  cache updating, 766-767
  delayed-write, 766
  group, 884
  mandatory, 826
  mechanisms vs., 668
  pageout, 800
  security, 653
  write-on-close, 766-767
  write-through, 766
**policy algorithm (Linux), 800**
**polling, 493-494**
**polymorphic viruses, 633**
**pools, 177-180, 483, 889**
**pop, 66**
**ports, 78, 129, 490**
  connection and communication, 138
  naming of, 135-136
  in remote procedure calls, 150
  well-known, 146
**portability, 834-835**
**Portable Executable (PE) format, 77**
**portals, 40**
**port driver, 864**
**port number, 746-747**
**port rights, 135**
**port scanning, 637**
**position-independent code (PIC), 803**
**positioning time (disks), 450**
**POSIX:**
  interprocess communication example, 132-135
  real-time scheduling, 232-234
  synchronization examples, 299-303
**POSIX 1e, 685, 686**
**possession (of capability), 680**
**POST (power-on self-test), 872**
**posting messages, 891**
**power framework (PoFX), 870**
**power management, 514-516**
**power manager (Windows 10), 870-871**
**power-of-2 allocator, 427**
**power-on self-test (POST), 872**
**power users, 60-61**
**PPTP (Point-to-Point Tunneling Protocol), 881**
**P + Q redundancy scheme, 478**
**preemptive kernels, 262**
**preemptive multitasking, 790**
**preemptive scheduling, 202-203**
**premaster secret (TLS), 647**
**prepaging, 430-431**
**presentation layer, 742**
**primary thread, 890**
**principle of least privilege, 626, 627, 668-669**
**priority (field) 243**
**priority-based scheduling, 229-230**
**priority-inheritance protocol, 284**
**priority inversion, 284, 285**
**priority number, 281**
**priority paging, 440**
**priority replacement algorithm, 420-421**
**priority scheduling algorithm, 211-214**
**private cloud, 44**
**private dispatch queues, 185**
**private keys, 641**
**privileged instructions, 25**
**privileged mode,** see **kernel mode**
**privilege escalation, 623**
**privilege separation, 669**
**procedural languages, 313**
**procedures, as domains, 674**

- **process(es), 23, 105-154**
  - aborting, 342
  - background, 74-75, 115, 123, 215, 241
  - browser, 124
  - communication between, see **interprocess communication**
  - components of, 106-107
  - consumer, 126-127, 290, 291, 559-560
  - context of, 788-789
  - cooperating, 123, 257
  - coordination among, 260
  - daemon, 690
  - defined, 103, 105
  - dispatched, 112
  - as domains, 674
  - empty, 123
  - environment of, 787-788
  - foreground, 115, 122, 215, 241
  - guest, 702
  - idle, 872
  - independent, 123
  - init, 117
  - I/O- vs. CPU-bound, 112
  - job vs., 106
  - lightweight, 193
  - in Linux, 789-790
  - multithreaded, see **multithreaded processes; multithreading**
  - operations on, 116-123
    - creation, 116-121
    - termination, 121-123
  - orphan, 122
  - parent, 111
  - passing data between, 813
  - periodic, 230
  - Pico, 91
  - plug-in, 124
  - producer, 126-127, 290, 558-559
  - renderer, 124
  - service, 123
  - sibling, 111
  - single-threaded, 160
  - state of, 107-109
  - system, 872
  - systemd, 117
  - threads performed by, 110
  - visible, 122
  - in Windows 10, 886
  - zombie, 122
- **process-contention scope (PCS), 217-218**
- **process control blocks (PCBs), 109-110**
- **process-control system calls, 66-71**
- **process identifier (pid), 116, 787**
- **process identity (Linux), 787**
- **Process Lifetime Manager (PLM), 837**
- **process management:**
  - about, 27-28
  - in Linux, 786-790
  - Windows 10, 886-891
- **process manager (Windows 10), 858-860**
- **process migration, 752-753**
- **process name, 73**
- **processors, 18**
  - distributed system, 734
  - front-end, 522
  - ideal, 242, 842
  - Intel, 379-382
    - event-vector table, 496
    - IA-32 architecture, 379-382
    - IA-64 architecture, 382
    - thread building blocks, 186-188
  - logical, 832
  - multi-, 18, 220
  - multicore, 221-224
- **processor affinity, 225-226**
- **processor groups, 832**
- **process reflection, 860**
- **process representation (Linux), 111**
- **process scheduler, 110-112**
- **process scheduling, 110-115, 199, 234**
- **process synchronization, 260.** See also **synchronization tools**
- **process termination, deadlock recovery by, 342**
- **/proc file system (Linux), 808-810**
- **procfs (file system), 598**
- **producer process, 126-127, 290, 558-559**
- **production kernels (Linux), 777**
- **program counters, 27, 106, 109**
- **program execution (operating system service), 56**
- **program loading and execution, 74**
- **programmable interval timer, 505**
- **programmed I/O (PIO), 498**
- **programmer interface (Windows 10), 884-895**
  - IPC with Windows messaging, 891-892
  - kernel object access, 884-885
  - memory management, 892-895
  - process management, 886-891
  - sharing objects between processes, 885-886
- **programming:**
  - multi-, 23, 112, 420
  - multicore, 162-166
- **programming-environment virtualization, 703, 717**
- **programming languages, 313-314**

**programming-language support, 74**
**program structure, for demand paging, 433-434**
**program threats, 625-634**
  code injection, 628-631
  malware, 625-628
  viruses, 631-634
  worms, 631, 632
**progress (requirement), 260**
**projects, 244**
**proportional allocation, 414-415**
**proportional share scheduling, 233**
**proprietary software, 46-47**
**protection, 667-698**
  access matrix model, 675-685
    implementation, 679-682
    mandatory access control, 684-685
    and revocation of access rights, 682-683
    role-based access control, 683-684
  capability-based systems, 685-687
  code signing, 690
  in computer systems, 33-34
  with contiguous memory allocation, 357
  domain of, 671-675
  file, 531
  file-system interface, 550-555
  goals of, 667-668
  I/O, 512
  language-based systems, 690-696
    compiler-based enforcement, 691-693
    in Java, 694-696
  as operating system service, 57-58
  in paged environment, 368-369
  and principle of least privilege, 668-669
  rings of, 669-671
  sandboxing, 689-690
  static vs. dynamic, 673
  system-call filtering, 688
  system integrity, 687-688
  from viruses, 657-659
**protection domains, 671-675, 711**
**protection mask (Linux), 817**
**protection rings, 25, 669-671**
**protection system calls, 73-74**
**pseudo-device driver, 721-722**
**PTBR (page-table base register), 365**
**PTEs (page-table entries), 853**
**PTE tables, 853**
**Pthreads, 169-171, 218-219**
**PTLR (page-table length register), 369**
**P-to-V conversion, 724**
**public cloud, 44**
**public domain software, 779-780**
**public keys, 641**
**public-key encryption, 641**
**pull migration, 224**
**pure code (reentrant), 370**
**pure demand paging, 395**
**pushing, 66, 519**
**Pushlocks, 831**
**push migration, 224**
**put command, 751**

**Q**

**Quest-V, 729**
**queue(s), 38**
  dispatch, 185
  main, 185
  ready, 112, 221, 843
  scheduling, 112-113
  wait, 112
**queuing diagram, 112, 113**
**queuing-network analysis, 247**

**R**

**race condition, 259, 261**
**RAID (redundant arrays of inexpensive disks), 473-485**
  extensions, 481-482
  levels of, 475-481
  for object storage, 483-485
  performance improvement, 475
  problems with, 482-483
  reliability improvement, 473-475
  structuring, 474
**RAID level 0, 476**
**RAID level 0 + 1, 478-479**
**RAID level 1, 476**
**RAID level 1 + 0, 478-479**
**RAID level 4, 476-477**
**RAID level 5, 477-478**
**RAID level 6, 478**
**RAID levels, 475-481**
  common, 475-478
  selecting, 480-481
  variations in, 478-480
**raising interrupts, 9, 494**
**RAM (random-access memory), 11**
**RAM drives, 454, 455**
**random-access devices, 502**
**random-access memory (RAM), 11**
**random-access time (disks), 450**
**range (value), 187**
**ransomware, 626**
**RAT (Remote Access Tool), 625**

- **rate-monotonic scheduling, 230-232**
- **rate-monotonic scheduling algorithm, 230-232**
- **raw disk, 413, 464, 601**
- **raw I/O, 464, 465, 503-504**
- **raw partitions, 468**
- **RBAC (role-based access control), 683-684**
- **RDP, 707**
- **reacquisition, of capabilities, 682**
- **read access, locks with, 292**
- **read-ahead technique, 585**
- **read-end (of pipe), 140**
- **readers, 291, 292**
- **readers-writers problem, 290-293**
- **reader-writer locks, 292-293**
- **reading files, 532, 551**
- **read-modify-write cycle, 477**
- **read only devices, 502**
- **read pointer, 532**
- **read-write devices, 502**
- **ready queues, 112, 221, 843**
- **ready state, 108, 109**
- **real-time class, 239**
- **real-time CPU scheduling, 227-234**
  - earliest-deadline-first scheduling, 232-233
  - and minimizing latency, 227-229
  - POSIX real-time scheduling, 233-234
  - priority-based scheduling, 229-230
  - proportional share scheduling, 233
  - rate-monotonic scheduling, 230-232
- **real-time embedded systems, 45-46**
- **real-time operating systems, 46**
- **real-time range (Linux schedulers), 790**
- **real-time scheduling (Linux), 792**
- **reapers, 417-418**
- **receives, blocking vs. nonblocking, 130**
- **reconfiguration, 755**
- **records:**
  - activation, 107
  - base file, 876
  - logical, 539
  - master boot, 465, 466
- **recovery:**
  - from deadlock, 341-343
  - from failure, 755-756
  - and file system implementation, 586-589
  - Windows 10, 877-878
- **recovery mode, 95**
- **red-black trees, 38, 40**
- **Red Hat, 779**
- **redirectors, 882-883**
- **redundancy, 473-475**
- **redundant arrays of inexpensive disks,** see **RAID**
- **reentrant code (pure code), 370**
- **reentrant locks, 307-308**
- **reference, locality of, 395**
- **reference bits, 409**
- **referenced pointer, 849**
- **reference string, 404, 406**
- **reflection process, 860**
- **reflector, 864**
- **regions, 383**
- **register(s):**
  - base, 351-352
  - control, 492
  - CPU, 110
  - data-in, 492
  - data-out, 492
  - instruction, 12
  - limit, 351-352
  - memory-address, 354
  - page-table base, 365
  - page-table length, 369
  - relocation, 354
  - status, 492
  - translation table base, 383
- **registry, 74, 871-872**
- **regression testing, 249**
- **relative access, 539-540**
- **relative block number, 540**
- **relative path names, 546**
- **release, of resources, 318**
- **reliability:**
  - of distributed systems, 735
  - RAID for improving, 473-475
  - of TCP, 748
  - of UDP, 747
  - of Windows 10, 828-829
- **relocatable code, 353**
- **relocatable object file, 75**
- **relocation, 75, 76**
- **relocation register, 354**
- **remainder section, 260**
- **Remote Access Tool (RAT), 625**
- **remote desktop, 874**
- **remote file access, 764-767**
- **remote file-systems, 605-608**
- **remote file transfer, 750-751**
- **remote login, 750**
- **remote operations (NFS), 615**
- **remote procedure calls (RPCs), 149-153, 834**
- **remote-service mechanism, 764**
- **removable storage media, 451**
- **remove() method, 305-307**

**renaming files 542**
**renderer processes, 124**
**rendezvous, 131**
**repair, mean time to, 474**
**replacement, page,** see **page replacement**
**replay attacks, 622**
**replication, 480, 592-593**
**reply port, 135**
**repositioning (in files) 532**
**Request (data structure), 335-336, 339, 340**
**requests, for resources, 318**
**request consumer (circular buffer), 716**
**request edge, 323**
**request manager, 811**
**request producer (circular buffer), 716**
**resident attributes, 876**
**resolution:**
  address resolution protocol, 745
  conflict, 784, 785
  of links, 548
  name, 738-741
  and page size, 431-432
**resource allocation (operating system service), 57**
**resource-allocation graph, 323-326, 334, 338**
**resource-allocation-graph algorithm, 333**
**resource allocator, operating system as, 5**
**resource arbiters, 869**
**resource management, 27-32**
**resource preemption, deadlock recovery by, 342-343**
**resource-request algorithm, 335**
**resource sharing, 162, 734-735**
**resource utilization, 4**
**responses (password), 652**
**response consumer (circular buffer), 716**
**response producer (circular buffer), 716**
**response time, 23, 205**
**responsibility, for run-time-based enforcement, 694**
**responsiveness, multithreaded process, 162**
**restart area, 878**
**restore, state, 114**
**restore point, system, 871**
**restoring data, 588-589**
**resuming, 717, 888**
**return from sleep, 243**
**reverse engineering, 47**
**revocation of access rights, 682-683**
**RHEL 7, 461**
**rich text format (RTF), 658**
**right child, 38**
**rights:**
  access, 534, 673, 680, 682-683
  copy, 677
  group, 817
  owner, 678, 817
  port, 135
  user, 817
  world, 817
**rings, protection, 669-671**
**risk assessment, 653-654**
**roaming profiles 883**
**robustness, distributed system, 754-756**
**roles, 683**
**role-based access control (RBAC), 683-684**
**rollback, 343**
**root directory, 877**
**rootkit viruses, 632**
**root partition, 601**
**rotational latency (disks), 451**
**rotations per minute (RPM), 450**
**round robin, 130**
**round-robin (RR) scheduling algorithm, 209-211**
**routers, 736**
**RPCs (remote procedure calls), 149-153, 834**
**RPM (rotations per minute), 450**
**RR scheduling algorithm, 209-211**
**RSA algorithm, 641, 642**
**RTE (run-time environment), 64-65**
**RTF (rich text format), 658**
**running state, 108, 109**
**running system, 94**
**run time, virtual, 236**
**run-time-based enforcement, 694-696**
**run-time environment (RTE), 64-65**

## S

**SaaS (software as a service), 44**
**safe computing, 658**
**safe sequence, 331**
**safe state, 331-333**
**safety, as virtualization requirement, 704**
**safety algorithm, 335**
**sandbox, 124, 658**
**sandboxing, 689-690**
**SANs,** see **storage-area networks**
**SAS buses, 456, 490**
**SATA buses, 456**
**save, 114, 592**
**scalability, 162, 484, 756-757**
**SCAN scheduling, 458-459**

**SCAN (elevator) scheduling algorithm, 458-459**
**scatter-gather method, 498, 508**
**schedulers:**
  CFQ, 461, 811
  Completely Fair, 236, 237, 790
  CPU, 113-114, 201
  deadline, 460, 461
  Linux, 790
  NOOP, 461
  process, 110-112
**scheduler activation, 192-194**
**scheduling:**
  cooperative, 202
  CPU, see **CPU scheduling**
  C-SCAN, 460
  earliest-deadline-first, 232-233
  fair, 791
  FCFS, 458, 459
  HDD, 457-461
  I/O, 508-509
  job, 106
  in Linux, 790-794
  multi-processor, see **multi-processor scheduling**
  nonpreemptive, 202
  NVM, 461-462
  preemptive, 202-203
  priority-based, 229-230
  process, 110-115, 199, 234
  proportional share, 233
  rate-monotonic, 230-232
  real-time, 792
  SCAN, 458-459
  selecting disk-scheduling algorithm, 460-461
  shortest-remaining-time-first, 209
  thread, 199, 790-791, 844-845
  user-mode, 241, 833, 890-891
  in Windows 10, 887
**scheduling classes, 236**
**scheduling context, 788**
**scheduling domain, 238**
**scheduling information, CPU, 110**
**scheduling rules, 887**
**SCM (service control manager), 870**
**scope:**
  contention, 217-218
  of lock, 305
**script kiddies, 631**
**scripts, shell, 61, 536**
**SCS (system-contention scope), 218**
**searching, for files, 541, 542**
**search path, 545**
**secondary memory, 395**
**secondary storage, 13.** See also **disk(s)**
  allocation of, 570-578
    contiguous allocation, 570-573
    indexed allocation, 575-577
    linked allocation, 573-575
    and performance, 578-579
  connection methods for, 456
**second-chance page-replacement algorithm (clock algorithm), 410-411**
**second extended file system (ext2), 805**
**second-level interrupt handler (SLIH), 496**
**second readers, 291**
**section objects, 139, 852**
**sectors, 450, 466, 566**
**sector slipping, 467**
**sector sparing, 466**
**Secure Boot, 872**
**secure by default, 634**
**secure kernel, 839-840**
**Secure Monitor Call (SMC), 670**
**secure shell, 116**
**secure system process, 872**
**secure systems, 622**
**Secure World, 838**
**security, 621-665.** See also **protection**
  of compiler-based enforcement, 692
  in computer systems, 33-34
  cryptography for, 637-648
    and encryption, 638-645
    implementation, 645-646
    TLS example, 646-648
  implementation of, 653-662
    and accounting, 659
    and auditing, 659
    and firewalling, 659-660
    and intrusion prevention, 655-657
    levels of defenses, 661-662
    and logging, 659
    and security policy, 653
    and virus protection, 657-659
    and vulnerability assessment, 653-655
  in Linux, 816-818
  as operating system service, 57-58
  as problem, 621-625
  and program threats, 625-634
    code injection, 628-631
    malware, 625-628
    viruses, 631-634
    worms, 631, 632
  and system/network threats, 634-637
  user authentication for, 648-653

  in Windows 10, 662-664, 826-828, 878
**security access tokens (Windows 10), 662**
**security context (Windows 10), 662-663**
**security descriptor (Windows 10), 663**
**security domains, 659**
**security ID (SID), 33, 867**
**security policy, 653**
**security reference monitor (SRM), 866-869**
**security-through-obscurity approach, 655**
**security tokens, 867**
**seek, file, 532**
**seek time (disks), 450**
**segmentation, IA-32, 379-380**
**selective revocation, 682**
**semantics, 510, 608-609**
**semaphore(s), 272-276**
  binary, 273
  counting, 273
  defined, 272
  dining-philosophers solution with, 294-295
  implementation, 274-276
  in Java, 308-309
  monitors using, 280-281
  named, 300-301
  POSIX examples, 300-302
  unnamed, 300-302
  usage of, 273-274
**semaphore objects (Windows 10), 845**
**semiconductor memory, 14**
**sends, blocking vs. nonblocking, 130**
**sending messages, 891**
**sense key, 512**
**separation hypervisors, 729**
**sequence numbers, 748**
**sequential access (files) 539, 541**
**sequential devices, 502**
**serial ATA (SATA) buses, 456**
**serial-attached SCSI (SAS) buses, 456, 490**
**serial dispatch queue, 185**
**server(s), 73**
  blade, 18-19
  bootstrap, 136
  in client-server model, 606, 758-759, 861-862
  defined, 757
  in distributed systems, 734
  file-server systems, 43
  name, 739
  and redirectors, 882-883
**server-initiated approach to verifying cached data, 767**
**Server Message Block (SMB), 880**
**server subject (Windows 10), 663**
**server systems, 42-43, 734, 874-875**
**service(s):**
  defined, 757
  denial of, 622, 636
  distributed naming, 607
  high-availability, 19
  infrastructure as, 44
  log-file, 878
  network information, 607
  operating system, 55-58, 74-75, 115, 152
  platform as, 44
  software as, 44
  theft of, 622
**service control manager (SCM), 870**
**service processes, 123**
**service-trigger mechanism, 870**
**session(s), 751, 874**
**session 0, 873**
**session hijacking, 623**
**session key, 647**
**session layer, 742**
**session manager subsystem (SMSS), 872-873**
**session semantics, 609**
**sets:**
  entry, 303, 304, 307
  hard working-set limits, 438
  of holes, 358
  SMT, 242
  wait, 304, 307
  working, 422-424, 438
**setuid attribute, 34**
**setuid bit, 674-675**
**SHA-1 message digest, 643**
**shadow copies, 863**
**sharable devices, 502**
**shares, 244**
**shared directories, 547**
**shared files 609**
**shared libraries, 356, 392**
**shared lock, 534**
**shared memory, 123, 125, 556-560**
**shared-memory model, 57, 73, 125-127, 132-136**
**shared ready queue, 843**
**shared system interconnect, 18**
**sharing:**
  file, 602-603
  information, 123
  load, 220
  and paging, 369-371
  resource, 162, 734-735
  space, 592
**sharing objects, 885-886**

**shells, 58, 116, 783**
**shell scripts, 61, 536**
**short duration locks, 272**
**shortest-job-first (SJF) scheduling algorithm, 207-209**
**shortest-next-CPU-burst algorithm, 207**
**shortest-remaining-time-first scheduling, 209**
**shoulder surfing 649**
**sibling process, 111**
**SID (security ID), 33, 867**
**Siemens Jailhouse project, 729**
**signals, 188-189, 812-813**
**signal-and-continue method, 279**
**signal-and-wait method, 279**
**signaled state, 297**
**signal handlers, 188-189**
**signal-handler table, 789**
**signatures, 633, 643, 656, 828**
**signature-based detection, 656**
**silos, 859**
**SIMD, 833**
**simple messages, 136**
**simple subject (Windows 10), 662**
**simulations, 248-249**
**simultaneous multithreading, 222**
**single indirect blocks, 576**
**single-level directories, 542-543**
**single-processor systems, 15-16**
**single step (mode), 72**
**single-threaded processes, 160**
**single-user mode, 95**
**singly linked lists, 37**
**SIP (System Integrity Protection), 687-688**
**Siri, 5**
**sites, distributed system, 734**
**64-bit computing, 383**
**SJF scheduling algorithm, 207-209**
**sketch, 70**
**slabs, 427-429, 797-798**
**slab allocation, 427-430, 797-798**
**Slackware, 779**
**sleep, return from, 243**
**SLIH (second-level interrupt handler), 496**
**slim reader-writer (SRW) locks, 889**
**SLOB allocator, 430**
**SLUB allocator, 430**
**small objects, 430**
**SMB (Server Message Block), 880**
**SMC (Secure Monitor Call), 670**
**SMP,** see **symmetric multiprocessing**
**SMSS (session manager subsystem), 872-873**
**SMT sets, 242**
**snapshots, 480, 588, 705, 879**
**sniffing 635-636, 649-650**
**social engineering, 624**
**sockets, 146-149**
**socket interface, 504**
**soft affinity, 225**
**soft errors, 463**
**soft page faults, 416**
**soft real-time systems, 227**
**software:**
  process migration and, 753
  proprietary, 46-47
  public domain, 779-780
**software as a service (SaaS), 44**
**software engineering, 80**
**software interrupts (traps), 497**
**software objects, 672**
**software transactional memory (STM), 312**
**Solaris, 51**
  file systems in, 482-484, 597, 599
  Oracle SPARC, 375-376
  scheduling example, 242-244
  virtual memory in, 438-440
  ZFS file system, 482-484, 581, 588, 598
**Solaris 10:**
  role-based access control in, 683, 684
  zones in, 718, 719
**solid-state disks (SSDs), 452**
**source-code viruses, 633**
**source files 530**
**space maps, 581**
**space sharing, 592**
**SPARC, 375-376**
**sparseness, 374, 391**
**special instructions, 709**
**special-purpose file systems, 597-598**
**specifications thread behavior, 169**
**speed of operations (I/O devices), 502**
**spinlocks, 272**
**split-screen, 115**
**spoofed identifiers 606**
**spoofing 636**
**spools, 511**
**spooling, 511**
**Springboard interface, 60, 87**
**spyware, 626**
**SRM (security reference monitor), 866-869**
**SRW (slim reader-writer) locks, 889**
**SSDs (solid-state disks), 452**
**stacks, 37-38, 66**
  device, 862
  LRU page replacement with, 408

  OSI protocol, 742-744
**stack algorithms, 408-409**
**stack inspection, 694, 695**
**stack section (of process), 107**
**stalling, 350**
**standard swapping, 377**
**standby page, 856**
**starvation (indefinite blocking), 213, 343**
**states:**
  application, 378
  new, 108
  nonsignaled vs. nonsignaled, 297
  of processes, 107-109
  ready, 108, 109
  running, 108, 109
  safe, 331-333
  suspended, 705, 888
  terminated, 109
  unsafe, 332-334
  waiting, 108, 109
**state information, 608**
**stateless DFS, 608**
**state restore, 114**
**state save, 114**
**static linking, 355-356, 803**
**static protection, 673**
**status information, 74**
**status register, 492**
**stealth viruses, 633**
**STM (software transactional memory), 312**
**storage, 11-14.** See also **mass-storage structure**
  cloud, 471, 751
  content-addressable, 484
  definitions and notations, 12
  host-attached, 470
  network-attached, 470-471
  nonvolatile, 14, 449
  object, 483-485
  secondary, 13, 456, 570-578. See also **disk[s]**
  tertiary, 13
  thread-local, 192, 894, 895
  utility, 481
  volatile, 11
**storage-area networks (SANs), 21, 470, 472**
**storage array, 472-473, 481**
**storage attachment, 469-473**
**storage devices, organization of, 597, 598**
**storage device management, 463-467**
**storage management, 30, 32, 723**
**stream ciphers, 640**
**stream head, 519**
**streaming transfer rate, 486**
**stream modules, 519**
**STREAMS mechanism, 519-521**
**string, reference, 404, 406**
**strongly ordered model, 265**
**strong passwords, 651**
**stubs, 150**
**subjects (Windows 10), 662-663**
**subsets, stack algorithm, 408**
**subsystems, 75**
**SunOS, 51**
**superblock, 566**
**superblock objects, 605, 804, 805**
**supervisor mode,** see **kernel mode**
**SuSE, 779**
**suspended state, 705, 888**
**swap map, 469**
**swapping, 113-114, 376-378**
  in Linux, 800
  on mobile systems, 377-378
  with paging, 377, 378
  standard, 377
  system performance under, 378
**swap space, 395, 468-469**
**swap-space management, 467-469**
**SwitchBranch mechanism, 830**
**switches, context, 114-115, 204**
**switching:**
  domain, 673, 674
  fast-user, 825, 874-875
**symbolic links, 879**
**symmetric clustering, 20**
**symmetric encryption, 639-640**
**symmetric encryption algorithm, 639**
**symmetric multiprocessing (SMP), 16, 220, 794**
**symmetry, in addressing, 129**
**synchronization, 130-131, 289-314**
  alternative approaches to, 311-314
  block, 305
  bounded-buffer problem, 290
  dining-philosophers problem, 293-295
  for interprocess communication, 812-813
  in Java, 303-311
    condition variables, 309-311
    monitors, 303-307
    reentrant locks, 307-308
    semaphores, 308-309
  kernel, 295-299, 792-794
  in Linux, 130-131, 812-813
  in message-passing model, 130-131
  in POSIX, 299-303
  process, 260. See also **synchronization tools**

  - readers-writers problem, 290-293
  - thread, 888-889
- **synchronization primitives, 845-846**
- **synchronization tools, 257-287**
  - about, 257-260
  - critical-section problem, 260-270
    - hardware solution to, 265-270
    - Peterson's solution to, 262-265
  - evaluation of, 284-286
  - and liveness, 283-284
  - monitors for, 276-282
    - resumption of processes within, 281-282
    - semaphores, implementation using, 280-281
    - usage, 277-280
  - mutex locks, 270-272
  - semaphores for, 272-276
- **synchronous devices, 502, 506, 507**
- **synchronous message passing, 130**
- **synchronous threading, 169**
- **synchronous writes, 585**
- **system administrators, 60**
- **system build, 92**
- **system calls (monitor calls), 22, 62-74**
  - and API, 63-66
  - clone(), 195-196
  - for communication, 72-73
  - for device management, 71-72
  - exec(), 188
  - for file management, 71
  - fork(), 188
  - functioning of, 62-63
  - for information maintenance, 72
  - for I/O, 512, 513
  - for process control, 66-71
  - for protection, 73-74
- **system-call filtering, 688**
- **system-call firewalls, 660**
- **system-call interface, 65**
- **system components (Windows 10), 838-874**
  - executive, 848-874
  - hardware-abstraction layer, 840
  - hyper-V hypervisor, 839
  - kernel, 840-848
  - secure kernel, 839-840
- **system-contention scope (SCS), 218**
- **system daemons, 22, 781**
- **system-development time, 705**
- **system disk, 465**
- **systemd process, 117**
- **system goals, 79**
- **System Integrity Protection (SIP), 687-688**
- **system libraries (Linux), 781**
- **system mode,** see **kernel mode**
- **system model, for deadlocks, 318-319**
- **system processes, 872**
- **system programs, 6**
- **system resource-allocation graph, 323-326**
- **system restore point, 871**
- **system utilities, 74-75, 781**
- **System V init, 117**
- **system-wide open-file table, 567**
- **system-wide tools, 96, 97**

T

- **table(s).** See also **page table(s)**
  - attribute-definition, 877
  - device-status, 508-509
  - event-vector, 11
  - file, 788
  - file-allocation, 574-575
  - frame, 365
  - global, 679
  - global descriptor, 379
  - handle, 849
  - hash, 570
  - master file, 566
  - mount, 517, 567
  - open-file, 533, 567
  - page directory pointer, 381
  - per-process open-file, 567
  - PTE, 853
  - signal-handler, 789
  - system-wide open-file, 567
- **tags, 680**
- **tapes, magnetic, 455**
- **target latency, 236, 791**
- **target thread, 190**
- **tasks, 106, 135, 195, 234.** See also **user programs (user tasks)**
- **task control blocks,** see **process control blocks**
- **task identification for multicore programming, 163**
- **task parallelism, 165, 166**
- **Task Self port, 135**
- **TB (terabyte), 12**
- **TBBs (thread building blocks), 186-188**
- **TCP (transmission control protocol), 743-749**
- **TCP/IP,** see **Transmission Control Protocol/Internet Protocol**
- **TCP sockets, 147**
- **TDI (Transport Driver Interface), 880**

**TEBs (thread environment blocks), 889-890**
**templating, 706**
**temporary revocation, 682**
**terabyte (TB), 12**
**terminal concentrators, 522**
**terminal server systems, 874-875**
**terminated state, 109**
**termination, 121-123, 342**
**tertiary storage devices, 13**
**testing, multicore programming in, 165**
**text files 530**
**text section (of process), 106**
**theft of service, 622**
**thin-client computing, 874-875**
**thin clients, 40**
**third extended file system (ext3), 805-807**
**32-byte memory, 363, 364**
**thrashing, 419-425**
  cause of, 419-422
  current practice, 425
  and page-fault-frequency strategy, 424-425
  and working-set model, 422-424
**threads, 159-197.** See also **threading**
  alertable, 846
  green, 167
  hardware, 222
  idle, 239, 842
  Java, 173-176
  kernel, 166, 217, 234
  kernel-mode, 841
  in Linux, 789-790
  and multicore programming, 162-166
  in operating systems, 194-196
  and process model, 110
  Pthreads, 169-171, 218-219
  scheduling of, 199
  target, 190
  user, 166, 217
  user-mode, 841
  in Windows 10, 841-845, 886-889, 894
**Thread attach, 860**
**thread building blocks (TBBs), 186-188**
**thread cancellation, 190-192**
**thread dumps, 339**
**thread environment blocks (TEBs), 889-890**
**threading:**
  asynchronous, 169
  hyper-, 222, 832
  implicit, 176-188
    fork join, 180-183
    Grand Central Dispatch, 185-186
    Intel thread building blocks, 186-188
    OpenMP and, 183-185
    thread pools and, 177-180
  issues:
    fork() and excel() system calls, 188
    scheduler activations, 192-194
    signal handling, 188-190
    thread cancellation, 190-192
    thread-local storage, 192
  multi-, 166-168, 222, 223
  synchronous, 169
**thread libraries, 168-176**
  about, 168-169
  Java, 173-176
  Pthreads, 169-171
  Windows, 171-173
**thread-local storage (TLS), 192, 894, 895**
**thread pools, 177-180, 889**
**thread scheduling, 199**
  in Linux, 790-791
  in Windows 10, 844-845
**threats, 622**
  program, 625-634
    code injection, 628-631
    malware, 625-628
    viruses, 631-634
    worms, 631, 632
  system/network, 634-637
**three-way handshake, 748**
**throughput, 204-205**
**thunking, 830**
**tightly coupled systems, 83**
**time:**
  compile, 352
  down, 572
  effective access, 397-398
  effective memory-access, 367
  execution, 353
  load, 353
  mean time between failures, 473, 474
  mean time of data loss, 474
  mean time to repair, 474
  positioning, 450
  random-access, 450
  response, 23, 205
  seek, 450
  system-development, 705
  turnaround, 205
  virtual run, 236
  waiting, 205
**time quantum, 209-211, 243**
**time quantum expired, 243**
**timers, 26-27, 505-506**
**timer objects, 845**

**time slice, 209-211, 790-791**
**timestamps, 531**
**timestamp counters (TSCs), 845**
**TLB,** see **translation look-aside buffer**
**TLB miss, 366**
**TLB reach, 432-433**
**TLB walk, 376**
**TLS (thread-local storage), 192, 894, 895**
**TLS (Transport Layer Security), 646-648**
**tmpfs (file system), 598**
**top half (interrupt service routines), 793-794**
**total revocation, 682**
**touch screens, 5**
**touch-screen interface, 56, 60**
**trace files, 248**
**tracing tools, 97-98**
**tracks, disk, 450**
**traditional computing, 40-41**
**traffic network, 635-636**
**transactions, 311, 587, 808**
**transactional memory, 311-312**
**transfer rates, 450, 451, 486**
**transition page, 856**
**translation:**
  binary, 708-710
  flash translation layer, 453-454
  network address, 723
  path-name, 614-615
**translation granules, 383**
**translation look-aside buffer (TLB), 365-368, 376, 384, 855**
**translation table base register, 383**
**transmission control protocol (TCP), 743-749**
**Transmission Control Protocol/Internet Protocol (TCP/IP), 36, 743-746, 880-881**
**transparency, 756, 761**
**Transport Driver Interface (TDI), 880**
**transport layer, 742**
**transport-layer protocol (TCP), 645**
**Transport Layer Security (TLS), 646-648**
**traps, 22, 89, 497, 847**
**trap-and-emulate method, 707-708**
**trap doors, 626, 627**
**traversing file system, 542**
**trees, 38, 39, 116**
**tree-structured directories, 545-547**
**TRIMing unused blocks, 581-582**
**trimming, automatic working-set, 438**
**triple DES, 639**
**triple indirect blocks, 576, 577**
**Trojan horses, 625-626**
**truncating files, 532**
**trusted addresses, 638**
**Trustlets, 838**
**TrustZone (TZ), 670, 671**
**TSCs (timestamp counters), 845**
**tunneling, attacks with, 659-660**
**turnaround time, 205**
**twisted pair cables, 736**
**two-factor authentication, 652**
**two-level directories, 543-545**
**two-level model, 168**
**two-level page-table scheme, 372-373**
**type 0 hypervisors, 702, 713-714**
**type 1 hypervisors, 703, 714-715**
**type 2 hypervisors, 703, 715-716**
**type safety (Java), 696**
**TZ (TrustZone), 670, 671**

U

**UDP (user datagram protocol), 743, 746-748**
**UDP sockets, 147**
**UEFI (Unified Extensible Firmware Interface), 94**
**UFD (user file directory), 543**
**UFS (UNIX file system), 565-566, 598**
**UI,** see **user interface**
**UMDF (User-Mode Driver Framework), 864**
**UMS,** see **user-mode scheduling**
**unbounded buffer, 126**
**unbounded capacity (of queue), 132**
**UNC (Uniform Naming Convention), 881**
**uncontended loads, 285**
**uncontended locks, 271**
**unified buffer cache, 583-585**
**Unified Extensible Firmware Interface (UEFI), 94**
**unified virtual memory, 583**
**Uniform Naming Convention (UNC), 881**
**unikernels, 728**
**universal serial buses (USBs), 456**
**Universal Windows Platform (UWP), 426**
**UNIX file system (UFS), 565-566, 598**
**UNIX operating system:**
  consistency semantics, 609
  inode, 577
  I/O kernel structure in, 513, 514
  permissions in, 553
  protection domain in, 674-675
  system calls, 68
  system structure, 82
**unloader, module, 783**

**unnamed data, 875**
**unnamed semaphores, 300-302**
**unsafe state, 332-334**
**unstructured data, 484**
**untrusted applet protection, 695**
**upcalls, 193**
**upcall handler, 193**
**updating policy, cache, 766-767**
**urgency value, 223**
**URL loader, 695**
**USB drive, 452**
**USBs (universal serial buses), 456**
**use, of resources, 318**
**users, 4-5, 603**
  as domains, 674
  multiple, file sharing between, 602-603
  other users (class), 551
  power, 60-61
**user accounts, 662**
**user authentication, 648-653**
**user control list, 561**
**user datagram protocol (UDP), 743, 746-748**
**user-defined signal handlers, 189**
**user experience layer (macOS and iOS), 87**
**user file directory (UFD), 543**
**user goals, 79**
**user IDs, 33, 531, 675**
**user-initiated class, 185-186**
**user-interactive class, 185**
**user interface (UI), 4-5, 56, 58-62**
**user mode, 24, 25, 782**
**User-Mode Driver Framework (UMDF), 864**
**user-mode scheduling (UMS), 241, 833, 890-891**
**user-mode threads (UT), 841**
**user programs (user tasks), 106, 353, 801-803**
**user rights (Linux), 817**
**user threads, 166, 217**
**UT (user-mode threads), 841**
**utility class, 186**
**utility storage, 481**
**UWP (Universal Windows Platform), 426**

## V

**VACB (virtual address control block), 865**
**VADs (virtual address descriptors), 857**
**valid-invalid bit, 368-369**
**valid page, 856**
**variables:**
  atomic, 269-270
  condition, 278, 279, 302-303, 309-311, 889
**variable class, 239**
**variable-partition schemes, 358**
**variable timer, 26**
**VCPU (virtual CPU), 707**
**vectored I/O, 507-508**
**verifier, 98**
**version control system, 49**
**vfork() (virtual memory fork), 400**
**VFS (virtual file system), 804-805**
**VFS layer, 601**
**victim, for resource preemption, 343**
**victim frames, 402**
**views, 557, 852**
**virtual address, 354**
**virtual address control block (VACB), 865**
**virtual address descriptors (VADs), 857**
**virtual address space, 390, 391, 799-800**
**VirtualBox project, 704**
**virtual CPU (VCPU), 707**
**virtual file-systems, 603-605**
**virtual file system (VFS), 804-805**
**virtual file system (VFS) layer, 601**
**virtualization, 34-35**
  defined, 701
  operating-system components for, 719-726
    CPU scheduling, 720
    I/O devices, 722-723
    live migration, 724-726
    memory management, 721-722
    storage management, 723
  para-, 703, 716-717
  programming-environment, 703, 717
  research, 728-729
**virtual machines, 34, 701-730.** See also **virtualization**
  benefits of, 704-707
  building blocks, 707-713
    binary translation, 708-710
    hardware assistance, 710-713
    trap-and-emulate method, 707-708
  examples, 726-728
  features of, 704-707
  history of, 703-704
  implementations, 713-719
    application containment, 718-719
    emulation, 717-718
    paravirtualization, 716-717
    programming-environment virtualization, 717
    type 0 hypervisors, 713-714
    type 1 hypervisors, 714-715

    - type 2 hypervisors, 715-716
    - and virtual machine life cycle, 713
  - life cycle, 713
- **virtual machine control structures (VMCSs), 711**
- **virtual machine managers (VMMs), 25-26, 35, 702**
- **virtual machine sprawl, 713**
- **virtual memory, 24, 389-441**
  - background on, 389-392
  - and copy-on-write technique, 399-401
  - demand paging for conserving, 392-399, 430-436
    - basic mechanism, 393-396
    - free-frame list, 396-397
    - with inverted page tables, 433
    - and I/O interlock, 434-436
    - and page size, 431-432
    - and performance, 397-399
    - and prepaging, 430-431
    - and program structure, 433-434
    - and TLB reach, 432-433
  - direct virtual memory access, 500
  - and frame allocation, 413-419
    - allocation algorithms, 414-415
    - global vs. local allocation, 415-418
    - minimum number of frames, 413-414
    - non-uniform memory access, 418-419
  - kernel, 801
  - and kernel memory allocation, 426-430
  - in Linux, 798-801
  - and memory compression, 425-426
  - network, 765
  - operating-system examples, 436-440
  - page replacement for conserving, 401-413
    - and application performance, 412-413
    - basic mechanism, 401-404
    - counting-based page replacement, 411-412
    - FIFO page replacement, 404-406
    - LRU-approximation page replacement, 409-411
    - LRU page replacement, 407-409
    - optimal page replacement, 406-407
    - and page-buffering algorithms, 412
  - and thrashing, 419-425
    - cause, 419-422
    - current practice, 425
    - page-fault-frequency strategy, 424-425
    - working-set model, 422-424
  - unified, 583
  - in Win32 API, 892, 893
- **virtual memory context, 789**
- **virtual memory fork, 400**
- **virtual memory (VM) manager, 851-858**
- **virtual memory regions, 799**
- **virtual private networks (VPNs), 646, 881**
- **virtual run time, 236**
- **virtual to physical (V-to-P) conversion, 724**
- **Virtual Trust Levels (VTLs), 838**
- **virus dropper, 632**
- **viruses, 631-634, 657-659**
- **virus signatures, 633**
- **visible processes, 122**
- **VMCSs (virtual machine control structures), 711**
- **VM manager, 851-858**
- **VMMs,** see **virtual machine managers**
- **VMware, 704, 726-727**
- **vnode, 604**
- **voice over IP (VoIP), 44**
- **voice recognition, 5**
- **volatile memory, 454-455**
- **volatile storage, 11**
- **volume, 464-465, 474**
- **volume control block, 566**
- **volume file, 876-877**
- **volume shadow copies, 879**
- **voluntary context switches, 204**
- **von Neumann architecture, 12**
- **VPNs (virtual private networks), 646, 881**
- **VSM Enclaves, 840**
- **VTLs (Virtual Trust Levels), 838**
- **V-to-P conversion, 724**
- **VT-x instructions, 710**
- **vulnerability assessment, 653-655**

## W

- **WAFL file system, 589-593**
- **wait-for graph, 337, 338**
- **waiting, busy, 272**
- **waiting state, 108, 109**
- **waiting time, 205**
- **wait() method, 305-307**
- **wait queue, 112**
- **wait set, 304, 307**
- **wait() system call, 119, 121-122**
- **WANs,** see **wide-area networks**
- **weakly ordered model, 265**
- **wear leveling, 454**
- **Web-distributed authoring and versioning (WebDAV), 881**
- **well-known ports, 146**
- **well-known port numbers, 747**

**wide-area networks (WANs), 36, 735, 737-738**
**WiFi (wireless) networks, 41, 736-737**
**Win32 API, 884-895**
  creating process, 119-120
  IPC with Windows messaging, 891-892
  kernel object access, 884-885
  memory management, 892-895
  process management, 886-891
  shared memory, 556-560
  sharing objects between processes, 885-886
**Windows operating system (generally):**
  anonymous pipes, 141, 145
  booting from storage device, 466
  interprocess communication example, 138-139
  scheduling example, 239-242
  synchronization within kernels, 296-298
  system calls, 68
  threads, 194-195
**Windows 7, 465, 822**
**Windows 8, 823**
**Windows 10, 821-896**
  access-control list management in, 555
  design principles, 826-838
    application compatibility, 830-831
    dynamic device support, 837-838
    energy efficiency, 836-837
    extensibility, 833-834
    international support, 835-836
    performance, 831-833
    portability of, 834-835
    reliability, 828-829
    security, 826-828
  developments, 823-825
  fast-user switching with, 874-875
  file system, 875-879
  history of, 821-825
  networking, 880-884
  programmer interface, 884-895
    IPC with Windows messaging, 891-892
    kernel object access, 884-885
    memory management, 892-895
    process management, 886-891
    sharing objects between processes, 885-886
  security in, 662-664
  system components, 838-874
    executive, 848-874
    hardware-abstraction layer, 840
    hyper-V hypervisor, 839
    kernel, 840-848
    secure kernel, 839-840
  terminal services, 874-875
  virtual memory in, 437-438
**Windows Desktop Bridge, 823**
**Windows Driver Foundation, 864**
**Windows executive, 848-874**
  booting, 872-874
  cache manager, 864-866
  client-server computing, 861-862
  I/O manager, 862-864
  object manager, 849-851
  plug-and-play manager, 869-870
  power manager, 870-871
  process manager, 858-860
  registry, 871-872
  security reference monitor, 866-869
  virtual memory manager, 851-858
**Windows group policy, 884**
**Windows messaging, 891-892**
**Windows Store, 823**
**Windows subsystem for Linux (WSL), 91**
**Windows Task Manager, 97**
**Windows thread library, 171-173**
**Windows Vista, 822**
**Windows XP, 822**
**WinRT, 823**
**Winsock, 891**
**wired down entries, 366**
**wireless access points, 736**
**wireless (WiFi) networks, 41, 736-737**
**word, 11**
**Work (data structure), 335, 339, 340**
**working sets, 422-424, 438**
**working-set maximum, 438**
**working-set minimum, 438**
**working-set model, 422-424**
**working-set window, 422**
**Workstation (VMWare), 726-727**
**work stealing algorithm, 182**
**world rights (Linux), 817**
**World Wide Web, 605, 737**
**worms, 631, 632**
**worst-fit strategy, 358, 359**
**writes, synchronous vs. asynchronous, 585**
**write access, locks with, 292**
**write amplification 462**
**write-anywhere file layout (WAFL) file system, 589-593**
**write-back caching, 766**
**write-end (of pipe), 140**
**write once devices, 502**
**write-on-close policy, 766-767**
**write pointer, 532**

writers, 291
write-through policy, 766
writing files, 532, 551
WSL (Windows subsystem for Linux), 91

X

x86-64 architecture, 382
XDR (external data representation), 150
Xen, 704, 716-717
XML firewalls, 660
Xtratum, 729

Y

yellow pages, 607

Z

zero capacity (of queue), 131
zero-day attacks, 656
zeroed page, 856
zero-fill-on-demand technique, 397
ZFS file system, 482-484, 581, 588, 598
zombie process, 122
zombie systems, 634, 635
zones, 718, 719, 795

# Glossary

**50-percent rule** A statistical finding that fragmentation may result in the loss of 50 percent of space.

**absolute code** Code with bindings to absolute memory addresses.

**absolute path name** A path name starting at the top of the file system hierarchy.

**abstract data type (ADT)** A programming construct that encapsulates data with a set of functions to operate on that data that are independent of any specific implementation of the ADT.

**access matrix** An abstract model of protection in which each row represents a domain, each column an object, and each entry a set of access rights.

**access right** The ability to execute an operation on an object.

**access-control list** A list of user names allowed to access a file.

**acknowledgment packet** In networking, a packet sent in response to the successful receipt of a message or packet.

**activation record** A record created when a function or subroutine is called; added to the stack by the call and removed when the call returns. Contains function parameters, local variables, and the return address.

**active directory (AD)** The Windows distributed information system, which is the Windows implementation of LDAP.

**acyclic graph** In directory structure implementation, a structure that contains no cycles (loops).

**adaptive mutex** A Solaris scheduling feature that starts as a standard spinlock and, if the object is locked and the lock is not held by a thread running on a CPU, blocks and sleeps until the lock is released.

**address space layout randomization (ASLR)** An operating system technique to avoid code-injection attacks that place memory objects like the stack and heap at unpredictable locations.

**address windowing extension (AWE)** A Windows mechanism for memory allocation that allows developers to directly request free pages of RAM from the memory manager and later commit virtual memory on top of those pages.

**address-space identifier** A part of a TLB entry that identifies the process associated with that entry and, if the requesting process doesn't match the ID, causes a TLB miss for address-space protection.

**address-space layout randomization (ASRL)** A Windows 7 feature that randomizes process memory addresses to avoid viruses that jump to specific code locations to gain privileges.

**admission control** In real-time scheduling, a practice whereby the scheduler may not allow a process to start if it cannot guarantee that the task will be serviced by its deadline.

**advanced configuration and power interface (ACPI)** Firmware common to PCs and servers that manages certain aspects of hardware, including power and device information.

**advanced encryption standard (AES)** The NIST cipher designed to replace DES and triple DES.

**advanced local procedure call (ALPC)** In Windows OS, a method used for communication between two processes on the same machine.

**advanced technology attachment (ATA)** An older-generation I/O bus.

**advisory file-lock mechanism** A file-locking system in which the operating system does not enforce locking and file access, leaving it to processes to implement the details.

**AFS (OpenAFS)** A network file system designed at Carnegie Mellon University with the goal of enabling servers to support as many clients as possible.

**aging** A solution to scheduling starvation that involves gradually increasing the priority of threads as they wait for CPU time.

**ahead-of-time (AOT) compilation** A feature of the Android RunTime (ART) virtual machine environment in which Java applications are compiled to native machine code when they are installed on a system (rather than just in time, when they are executed).

**allocation problem** The determination by the operating system of where to store the blocks of a file.

**Amazon Elastic Compute Cloud (ec2)** An instance of cloud computing implemented by Amazon.

**AMD 64** A 64-bit CPU designed by Advanced Micro Devices; part of a class of CPUs collectively known as x86-64.

**AMD-V** AMD CPU virtualization technology.

**analytic evaluation** A means of comparing scheduling-algorithm effectiveness by analyzing an algorithm against a workload and assigning it a score.

**anomaly detection** In intrusion detection, the use of various techniques to detect anomalous behavior that could be a sign of an attack.

**anonymous access** Remote access that allows a user to transfer files without having an account on the remote system.

**anonymous memory** Memory not associated with a file. Pages not associated with a file, if dirty and paged out, must not lose their contents and are stored in swap space as anonymous memory.

**Apple file system (APFS)** The 2017 file system from Apple that is the default on all modern Apple devices; features a rich feature set, space sharing, clones, snapshots, and copy-on-write design.

**Apple iOS** The mobile operating system created by Apple Inc.

**application component** In Android, a basic building block that provides utility to an Android app.

**application containment** A virtualization-like operating system feature that segregates applications from the operating system (examples include Solaris Zones, BSD Jails, and IBM WPARs).

**application frameworks layer** In the layered macOS and iOS operating system design, the layer that that includes Cocoa and Cocoa Touch frameworks, providing an API for Objective-C and Swift programming languages.

**application programming interface (API)** A set of commands, functions, and other tools that can be used by a programmer in developing a program.

application program A program designed for end-user execution, such as a word processor, spreadsheet, compiler, or Web browser.

**application proxy firewall** A firewall that understands protocols spoken by applications across a network, accepts connections to a target, and creates connections to that target, limiting and fixing what is sent from the originator.

**application state** A software construct used for data storage.

**arbitrary code guard (ACG)** A Windows 7 exploit-mitigation feature.

**argument vector** In Linux and UNIX, a list containing the command-line arguments used to invoke a process (and available to the process).

**ASIC** An application-specific integrated circuit (hardware chip) that performs its tasks without an operating system.

**assignment edge** In a system resource-allocation graph, an edge (arrow) indicating a resource assignment.

**asymmetric clustering** A configuration in which one machine in a cluster is in hot-standby mode while the other is running applications.

**asymmetric encryption algorithm** A cipher algorithm in which different keys are used for encryption and decryption.

**asymmetric multiprocessing** A simple multiprocessor scheduling algorithm in which only one processor accesses the system data structures and others run user threads, reducing the need for data sharing. A boss processor controls the system; the other processors either look to the boss for instruction or have predefined tasks.

**asynchronous** In I/O, a request that executes while the caller continues execution.

**asynchronous procedure call (APC)** A facility that enables a user thread to specify a function that is to be called when the user thread receives notification of a particular event.

**asynchronous threading** Threading in which a parent creates a child thread and then resumes execution, so that the parent and child execute concurrently and independently of one another.

**asynchronous write** A write that is buffered and written in arbitrary order, with the requesting thread continuing execution after the write is requested.

**atomic safe-save** In APFS, a feature primitive that performs file, bundle of files, and directory renaming in single atomic operations.

**atomic variable** A programming language construct that provides atomic operations on basic data types such as integers and booleans.

**atomically** A computer activity (such as a CPU instruction) that operates as one uninterruptable unit.

**attack** An attempt to break a computer system's security.

**attack surface** The sum of the methods available to attack a system (e.g., all of the network ports that are open, plus physical access).

**attacker** Someone attempting to breach a computer system's security.

**attribute** In the Windows NTFS file system, one of the elements making up a file. Each file is seen as a structured object consisting of typed attributes, with each attribute an independent byte stream that can be created, deleted, read, and written.

**audit trail** The collection of activities in a log for monitoring or review.

**authentication** The process of correctly identifying a person or device. In cryptography, constraining the set of potential senders of a message.

**automatic working-set trimming** In Windows, a process whereby, if a threshold of minimum free memory is reached, the number of working-set frames is decreased for every process.

**automount** A network/distributed file system feature in which file systems from remote servers are automatically located and mounted as needed.

**autoprobe** In Linux, a device driver probe that auto-detects device configuration.

**B+ tree** A tree data structure in which every path from the root of the tree to the leaf is the same length.

**back door** A daemon left behind after a successful attack to allow continued access by the attacker. In cryptography, a method of gaining access to encrypted information without first having the secret keys. More generally, a method of passing arbitrary commands or information when an interface does not provide a standard method.

**background** Describes a process or thread that is not currently interactive (has no interactive input directed to it), such as one not currently being used by a user. In the Grand Central Dispatch Apple OS scheduler, the scheduling class representing tasks that are not time sensitive and are not visible to the user.

**backing store** The secondary storage area used for process swapping.

**backup** In file systems, a copy or copies of the file system or changes to the file system that can be used to restore the file system if it is damaged or destroyed.

**bad block** An unusable sector on an HDD.

**balanced binary search tree** A tree containing n items that has, at most, lg n levels, thus ensuring worst-case performance of O(lg n).

**bandwidth** The total amount of data transferred divided by the total time between the first request for service and the completion of the last transfer.

**banker's algorithm** A deadlock avoidance algorithm, less efficient than the resource-allocation graph scheme but able to deal with multiple instances of each resource type.

**base file record** In the Windows NTFS file system, a descriptor of a large file containing pointers to overflow records that hold additional pointers and attributes.

**base register** A CPU register containing the starting address of an address space. Together with the limit register, it defines the logical address space.

**basic file system** A logical layer of the operating system responsible for issuing generic commands to the I/O control layer, such as "read block x," and also buffering and caching I/O.

**batch interface** A method for giving commands to a computer in which commands are entered into files, and the files are executed, without any human interaction.

**Bayes' theorem** A formula for determining conditional probability—e.g., the probability of an intrusion record detecting a real intrusion.

**Belady's anomaly** An anomaly in frame-allocation algorithms in which a page-fault rate may increase as the number of allocated frames increases.

**best-fit** In memory allocation, selecting the smallest hole large enough to satisfy the memory request.

**big data** Extremely large sets of data; distributed systems are well suited to working with big data.

**big.LITTLE** ARM processor implementation of HMP in which high-performance big cores are combined with energy efficient LITTLE cores.

**big-endian** A system architecture in which the most significant byte in a sequence of bytes is stored first.

**binary search tree** A type of binary tree data structure that requires an ordering between the parent's two children in which left child ⇐ right child.

**binary semaphore** A semaphore of values 0 and 1 that limits access to one resource (acting similarly to a mutex lock).

**binary translation** A virtualization method in which, when a guest is running in virtual kernel mode, every instruction is inspected by the virtual machine manager, and instructions that would behave differently in real kernel mode are translated into a set of new instructions that perform the equivalent task; used to implement virtualization on systems lacking hardware support for virtualization.

**binary tree** A tree data structure in which a parent may have at most two children.

**binder** In Android RPC, a framework (system component) for developing object-oriented OS services and allowing them to communicate.

**bind** Tie together. For example, a compiler binds a symbolic address to a relocatable address so the routine or variable can be found during execution.

**Bionic** A type of standard C library used by Android; it has a smaller memory footprint than glibc and is more efficient on slower (mobile) CPUs.

**BIOS** Code stored in firmware and run at boot time to start system operation.

**bit** The basic unit of computer storage. A bit can contain one of two values, 0 or 1.

**bit vector** A string of n binary digits that can be used to represent the status of n items. The availability of each item is indicated by the value of a binary digit: 0 means that the resource is available, while 1 indicates that it is unavailable (or vice-versa).

**bit-level striping** The splitting of data at the bit level, with each bit in a byte or word stored on a separate device.

**bitmap** A string of n binary digits that can be used to represent the status of n items. The availability of each item is indicated by the value of a binary digit: 0 means that the resource is available, while 1 indicates that it is unavailable (or vice-versa).

**BKL** In Linux kernel version 2.2, the “big kernel lock” spinlock, which protected all kernel data structures.

**blade server** A computer with multiple processor boards, I/O boards, and networking boards placed in the same chassis. The difference between these and traditional multiprocessor systems is that each blade-processor board boots independently and runs its own operating system.

**block** A self-contained unit of work. The smallest physical storage device storage unit, typically 512B or 4KB. In the Grand Central Dispatch Apple OS scheduler, a language extension that allows designation of a section of code that can be submitted to dispatch queues.

**block cipher** A cipher that works on blocks of data (rather than bits).

**block device** An I/O device that is randomly accessible, with block-size chunks being the smallest I/O unit.

**block-device interface** The interface for I/O to block devices.

**blocking** In interprocess communication, a mode of communication in which the sending process is blocked until the message is received by the receiving process or by a mailbox and the receiver blocks until a message is available. In I/O, a request that does not return until the I/O completes.

**block-level striping** The splitting of data at the block level, with each block stored on a separate device.

**boot block** A block of code stored in a specific location on disk with the instructions to boot the kernel stored on that disk. The UFS boot control block.

**boot control block** A storage block of data containing information needed by the system to boot from the volume containing the block.

**boot disk** A disk that has a boot partition and a kernel to load to boot the system. A device that has a boot partition and can store an operating system for booting the computer.

**boot partition** A storage device partition containing an executable operating system.

**boot sector** The first sector of a Windows boot device, containing the bootstrap code.

**booting** The procedure of starting a computer by loading the kernel.

**bootstrap** The set of steps taken at computer power-on to bring the system to full operation.

**bootstrap loader** The small program that loads the kernel as part of the bootstrap procedure.

**bootstrap port** In Mach message passing, a predefined port that allows a task to register a port it has created.

**bootstrap program** The program that allows the computer to start running by initializing hardware and loading the kernel.

**bootstrap server** In Mach message passing, a system-wide service for registering ports.

**Border Gateway Protocol** A network protocol for determining network routes so that packets can move from source to destination across a WAN.

**bottleneck** A performance-limiting aspect of computing (e.g., poorly written code or a hardware component that is not as fast as others in the system).

**bounded buffer** A buffer with a fixed size.

**bounded waiting** A practice that places limits on the time a thread or process is forced to wait for something.

**Bourne-Again shell** A common shell, or command interpreter.

**BPF compiler collection (BCC)** A rich toolkit for tracing system activity on Linux for debugging and performance-tuning purposes.

**bridging** In networking, the connection of two devices—e.g., a virtual-machine guest and the network to which the host computer is connected.

**broadcast** In networking, the sending of one or more packets to all devices on the local network.

**browser** A process that accepts input in the form of a URL (Uniform Resource Locator), or web address, and displays its contents on a screen.

**buddies** Pairs of equal size, used in the buddy method of memory allocation.

**buddy heap** In Linux, the use of a power-of-two allocator for the kernel heap.

**buffer** A memory area that stores data being transferred (e.g., between two devices or between a device and a process).

**buffer cache** In file I/O, a cache of blocks used to decrease device I/O.

**bug** An error in computer software or hardware.

**bus** A communication system; e.g., within a computer, a bus connects various components, such as the CPU and I/O devices, allowing them to transfer data and commands.

**busy waiting** A practice that allows a thread or process to use CPU time continuously while waiting for something. An I/O loop in which an I/O thread continuously reads status information while waiting for I/O to complete.

**byte** Eight bits.

**bytecode** A computer object code resulting from compiling a program in a language (such as Java) that runs on a virtual machine.

**C library (libc)** The standard UNIX/Linux system API for programs written in the C programming language.

**cache** A temporary copy of data stored in a reserved memory area to improve performance. In the slab allocator, a cache consists of two or more slabs.

**cache coherency** The coordination of the contents of caches such that an update to a value stored in one cache is immediately reflected in all other caches that hold that value.

**cache management** The management of a cache's contents.

**cache-consistency problem** A challenge in caching in which the cached copies of data must be kept consistent with the master copy of the data.

**caching** The use of temporary data storage areas to improve performance.

**cancellation point** With deferred thread cancellation, a point in the code at which it is safe to terminate the thread.

**Canonical** A popular Linux distribution.

**capability** In protection, the representation of an object in a capability list.

**capability list** In protection, a list of objects together with the operations allowed on those objects.

**capability-based protection** A protection facility in which the powers of root are divided into specific abilities, each represented by a bit in a bit mask that is used to allow or deny operations.

**cascading termination** A technique in which, when a process is ended, all of its children are ended as well.

**Ceph** A brand of object storage management software.

**certificate authority** A trusted signer of digital certificates.

**character device** An I/O device that has a character (byte) as its smallest I/O unit.

**character-stream interface** The interface for I/O to character devices (like keyboards).

**checksum** The general term for an error detection and correction code.

**children** In a tree data structure, nodes connected below another node.

**chip multithreading (CMT)** A CPU with multiple cores, where each core supports multiple hardware threads.

**chipset** The CPU and support chips that create a computer and define its architecture.

**circular buffer** A buffer that uses a fixed amount of storage and wraps writes from the end of the buffer to the beginning when the buffer fills (so that the buffer acts as if it's circular in shape).

**Circular SCAN (CSCAN) scheduling** An HDD I/O scheduling algorithm in which the disk head moves from one end of the disk to the other performing I/O as the head passes the desired cylinders; the head then reverses direction and continues.

**claim edge** In the deadlock resource-allocation-graph algorithm, an edge indicating that a process might claim a resource in the future.

**class loader** In Java, a helper component that loads .class files for execution by the Java virtual machine.

**class** In Java, a program component that is a collection of data fields and functions (methods) that operate on those fields.

**clean-up handler** A function that allows any resources a thread has acquired to be released before the thread is terminated.

**client** A computer that uses services from other computers (such as a web client). The source of a communication.

**client interface** In distributed computing, a set of services provided to a caller of services.

**client system** A computer that uses services from other computers (such as a web client).

**client-server model** A mode of computing in which a server provides services to one or more clients. In distributed computing, a model in which a computer acts as a resource server to other computers that are clients of those resources.

**client-side caching (CSC)** In Windows, a caching method used to allow remote users to work off-line and then consolidate their changes once they are online.

**clock** In the second-chance page-replacement algorithm, a circular queue that contains possible victim frames. A frame is replaced only if it has not been recently referenced.

**clone** In file systems, a snapshot that is read-write and modifiable. In virtualization, a copy of a guest that enables another instance of the guest to run in a separate virtual machine.

**CLOOK scheduling** An HDD I/O scheduling algorithm that modifies CSCAN by stopping the head after the final request is completed (rather than at the innermost or outermost cylinder).

**closed-source** An operating system or other program available only in compiled binary code format.

**closure** In functional programming languages, a construct to provide a simple syntax for parallel applications.

**cloud computing** A computing environment in which hardware, software, or other resources are made available to customers across a WAN, such as the Internet, usually with APIs for management. A type of computing that delivers computing, storage, and even applications "as a service" across a network.

**cloud storage** Storage accessed from a computer over a network to a distant, shared resource data center.

**cluster** In Windows storage, a power-of-2 number of disk sectors collected for I/O optimization.

**cluster-based model** In distributed computing, a model of resource sharing where all systems are equal users and providers of resources.

**clustered file system (CFS)** A file system that is LAN-based and treats N systems storing data and Y clients accessing the data as a single client-server instance; more complex than a client-server DFS but less complex than a cluster-based DFS. GPFS and Lustre are examples.

**clustered page table** A page table similar to a hashed page table, but a table entry refers to several (a cluster of) pages.

**clustered system** A system that gathers together multiple CPUs. Clustered systems differ from multiprocessor systems in that they are composed of two or more individual systems—or nodes—joined together.

**clustering** In general, gathering N items together. In virtual memory, paging in a group of contiguous pages when a single page is requested via a page fault.

**cluster** In file system block allocation, several contiguous blocks.

**coalescing** In general, combining. In the buddy memory allocation algorithm, freed memory in adjacent buddies can be coalesced into larger segments.

**code integrity** A Windows 7 module that checks the digital signatures of kernel modules to be sure they have not been tampered with by attackers.

**code review** A software development method in which the developer submits code to other developers for review and approval.

**code signing** The use of a digital signature to authenticate a program.

**code-injection attack** An attack that modifies otherwise well-behaved executable code.

**command interpreter** The operating system component that interprets user commands and causes actions based on them.

**command-line interface (CLI)** A method of giving commands to a computer based on a text input device (such as a keyboard).

**Common Criteria** The international 2005 successor to the Orange Book standard developed by the U.S. Department of Defense.

**common Internet file system (CIFS)** The Windows network file system, now used on many systems.

**communication port** In Windows OS, a port used to send messages between two processes.

**communications** A category of system calls.

**compaction** The shuffling of storage to consolidate used space and leave one or more large holes available for more efficient allocation.

**compartmentalization** The process of protecting each system component through the use of specific permissions and access restrictions.

**Completely Fair Queuing (CFQ)** In Linux, the default I/O scheduler in kernel 2.6 and later versions.

**Completely Fair Scheduler (CFS)** In Linux, the priority-based, preemptive scheduler included with the 2.6 kernel.

**component object model (COM)** The Windows mechanism for interprocess communication.

**compression** The use of algorithms to reduce the amount of data stored or sent.

**compression ratio** In memory compression, a measurement of the effectiveness of the compression (the ratio of the compressed space to the original amount of uncompressed space).

**compression unit** In NTFS, a unit of 16 contiguous clusters used in memory compression.

**computation migration** The use of a network to allow a task to remotely request resources to speed a computation.

**computation speedup** An increase in the amount of CPU compute power.

**computational kernel** A Windows mechanism for specifying tasks to run on GPUs.

**compute-server system** A server that provides an interface to which a client can send a request for an action (e.g., read data). In response, the server executes the action and sends the results to the client.

**Concurrency Runtime (ConcRT)** A Microsoft Windows concurrent programming framework for C++ that is designed for task-based parallelism on multicore processors.

**condition variable** A component of a monitor lock; a container of threads waiting for a condition to be true to enter the critical section.

**conditional-wait** A component of the monitor construct that allows for waiting on a variable with a priority number to indicate which process should get the lock next.

**confinement problem** The problem of guaranteeing that no information initially held in an object can migrate outside of its execution environment.

**conflict phase** During scheduling, the time the dispatcher spends moving a thread off a CPU and releasing resources held by lower-priority threads that are needed by the higher-priority thread that is about to be put onto the CPU.

**conflict-resolution mechanism** In Linux, the kernel module facility that allows different device drivers to reserve hardware resources and to protect those resources from accidental use by other drivers.

**congestion control** In networking, the attempt to approximate the state of networks between senders and receivers to avoid packet loss.

**connection port** In Windows OS, a communications port used to maintain connection between two processes, published by a server process.

**connectionless socket (UDP)** In Java, a mode of communication.

**connection-oriented socket (TCP)** In Java, a mode of communication.

**consistency checker** A system program, such as UNIX fsck, that reads file system metadata and tries to correct any inconsistencies (such as a file's length being incorrectly recorded).

**consolidation** In virtualization, the practice of running multiple guests per host, reducing the number of physical servers needed for a given workload.

**constant angular velocity (CAV)** A device-recording method in which the medium spins at a constant velocity and the bit density decreases from inner to outer tracks.

**constant linear velocity (CLV)** A device-recording method that keeps a constant density of bits per track by varying the rotational speed of the medium.

**consumer** A process role in which the process consumes information produced by a producer process.

**container** In application containment, a virtual layer between the operating system and a process in which the application runs, limiting its normal access to system resources.

**container object** In Windows 10, a category of objects that contain other objects (e.g., directories, which contain files).

**containers** In APFS, large free spaces in storage from which file systems can draw allocations.

**containment** A method for preventing deadlocks by creation and lock management of a higher-order resource that contains the locks of lower-order resources.

**contended** A term describing the condition of a lock when a thread blocks while trying to acquire it.

**content addressable storage** Another term for object storage; so called because objects can be retrieved based on their contents.

**context** When describing a process, the state of its execution, including the contents of registers, its program counter, and its memory context, including its stack and heap.

**context switch** The switching of the CPU from one process or thread to another; requires performing a state save of the current process or thread and a state restore of the other.

**contiguous allocation** A file-block allocation method in which all blocks of the file are allocated as a contiguous chunk on secondary storage.

**contiguous bit** In ARM v8 CPUs, a TLB bit indicating that the TLB holds a mapping to contiguous blocks of memory.

**contiguous memory allocation** A memory allocation method in which each process is contained in a single section of memory that is contiguous to the section containing the next process.

**control partition** In type 0 virtualization, a virtual hardware partition that provides services to the other partitions (and has higher privileges).

**control program** A program that manages the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices.

**control register** A device I/O register where commands are placed by the computer.

**control-flow guard (CFG)** A Windows 7 exploit-mitigation feature.

**controller** A special processor that manages I/O devices.

**convoy effect** A scheduling phenomenon in which a number of threads wait for one thread to get off a core, causing overall device and CPU utilization to be suboptimal.

**cooperating process** A process that can affect or be affected by other processes executing in the system.

**cooperative** A form of scheduling in which threads voluntarily move from the running state.

**coordination** Ordering of the access to data by multiple threads or processes.

**copy protection** The implementation of a mechanism to prevent the unauthorized copying of a licensed work (e.g., media, programs, operating systems).

**copy semantics** The meaning assigned to data copying—e.g., whether a block write from a process allows the data to be modified after the write has been requested.

**copy-on-write** Generally, the practice by which any write causes the data to first be copied and then modified, rather than overwritten. In virtual memory, on a write attempt to a shared page, the page is first copied, and the write is made to that copy.

**core** Within a CPU, the component that executes instructions.

**core dump** A copy of the state of a process written to disk when a process crashes; used for debugging.

**core frameworks** In the layered macOS and iOS operating system design, the layer that defines frameworks that support graphics and media, including QuickTime and OpenGL.

**counting semaphore** A semaphore that has a value between 0 and N, to control access to a resource with N instances.

**CPU burst** Scheduling process state in which the process executes on CPU.

**CPU scheduler** Kernel routine that selects a thread from the threads that are ready to execute and allocates a core to that thread.

**CPU scheduling** The process by which the system chooses which job will run next If several jobs are ready to run at the same time.

**CPU-bound process** A process that spends more time executing on CPU than it does performing I/O.

**crash** Termination of execution due to a problem. A failure in the kernel results in a system crash and reboot, while a process failure results in a process crash.

**crash dump** A copy of the state of the kernel written to disk during a crash; used for debugging.

**critical section** A section of code responsible for changing data that must only be executed by one thread or process at a time to avoid a race condition.

**critical-section object** A user-mode mutex object that can often be acquired and released without kernel intervention; a Windows OS scheduling feature.

**cryptography** A tool used to constrain the potential senders and/or receivers of a message (or stored data).

**current-file-position pointer** A per-process pointer to the location in a file to which the next read or from which the next write will occur.

**cycle** A repeating loop.

**cycle stealing** The act of a device, such as a DMA controller, using the bus and preventing the CPU from using it temporarily.

**cylinder** On an HDD, the set of tracks under the read-write heads on all platters in the device.

**cylinder groups** A sequential collection of storage media (typically a group of cylinders) that are managed as a single unit and subdivided into blocks.

**daemon** A service that is provided outside of the kernel by system programs that are loaded into memory at boot time and run continuously.

**daisy chain** In computer I/O, a connection method involving connecting devices to each other in a string (device A to B, B to C, C to D, etc.).

**dark web** The part of the World Wide Web that is not easy to reach (say, by search engines) and that is sometimes used for bad behavior (such as selling information stolen in successful attacks).

**Darwin** The Apple code name for its open-source kernel.

**data execution prevention (DEP)** A Windows 7 exploit-mitigation feature.

**data migration** The entire transfer of one or more files from one site to another.

**data parallelism** A computing method that distributes subsets of the same data across multiple cores and performs the same operation on each core.

**data section** The data part of a program or process; it contains global variables.

**data striping** The splitting of data across multiple devices.

**data-encryption standard (DES)** A cipher (algorithm for doing encryption and decryption) provided by the U.S. National Institute of Standards and Technology (NIST).

**datagram** A basic transfer unit on a packet-switched network like the Internet protocol (i.e., a network packet).

**data-in register** A device I/O register where data is placed to be sent to the device.

**data-out register** A device I/O register where data is placed by the device to be read by the computer.

**DCOM** The distributed computing extension to object linking and embedding (OLE).

**deadlock** The state in which two processes or threads are stuck waiting for an event that can only be caused by one of the processes or threads.

**deadlock avoidance** An operating system method in which processes inform the operating system of which resources they will use during their lifetimes so the system can approve or deny requests to avoid deadlock.

**deadlock prevention** A set of methods intended to ensure that at least one of the necessary conditions for deadlock cannot hold.

**deadlocked** The state in which two processes or threads are stuck waiting for an event that can only be caused by one of the processes or threads.

**Debian** A popular Linux distribution.

**debugger** A system program designed to aid programmers in finding and correcting errors.

**debugging** The activity of finding and removing errors.

**deduplication** The removal of duplicate information (bits, blocks, or files).

**default heap** The heap data structure created when a Win32 process is initialized.

**default signal handler** The signal handler that receives signals unless a user-defined signal handler is provided by a process.

**defense in depth** The theory that more layers of defense provide stronger defense than fewer layers.

**deferred procedure call (DPC)** In Windows scheduling, a call initiated by the interrupt that occurs when a time quantum expires, eventually causing the expired thread to be moved off a core and replaced with the next thread in the ready queue.

**degree of multiprogramming** The number of processes in memory.

**delayed-write policy** In caching, a policy whereby data are first written to a cache; later, the cache writes the change to the master copy of the data.

**demand paging** In memory management, bringing in pages from storage as needed rather than, e.g., in their entirety at process load time.

**demand-zero memory** In Linux, a virtual memory region backed by nothing; when a process tries to read a page in such a region, it is given a page of memory filled with zeros.

**demilitarized zone** In firewalling, a security domain less trusted than some other security domain (e.g., the domain containing a web server compared to the domain containing the crucial company database).

**denial-of-service** Preventing legitimate use of a system.

**dentry object** The VFS representation of a directory.

**desktop** In a GUI, the standard workspace represented by the GUI on the screen, in which a user executes tasks.

**desktop activity moderator (DAM)** A Windows 8 component that supports the system state “connected standby” on mobile devices, freezing computer activity but allowing rapid return to full functionality.

**desktop window manager** A Windows Vista user-mode component to manage GUI windows.

**deterministic modeling** A type of analytic evaluation that takes a particular predetermined workload and defines the performance of each algorithm for that workload.

**development kernel** A kernel released for developers to use, rather than for production use.

**device controller** The I/O managing processor within a device.

**device driver** An operating system component that provides uniform access to various devices and manages I/O to those devices.

**device manipulation** A category of system calls.

**device object** In Windows, the object representing a device.

**device queue** The list of processes waiting for a particular I/O device.

**device-status table** A kernel data structure for tracking the status and queues of operations for devices.

**digital certificate** A public key digitally signed by a trusted party.

**digital rights management** The implementation of a mechanism to prevent the unauthorized copying of a licensed work (e.g., media, programs, operating systems).

**digital signature** The authenticator produced by a digital-signature algorithm.

**digital-signature algorithm** A cryptographic checksum calculated in asymmetric encryption; used to authenticate a message.

**dining-philosophers problem** A classic synchronization problem in which multiple operators (philosophers) try to access multiple items (chopsticks) simultaneously.

**direct access** A file-access method in which contents are read in random order, or at least not sequentially.

**direct blocks** In UFS, blocks containing pointers to data blocks.

**direct communication** In interprocess communication, a communication mode in which each process that wants to communicate must explicitly name the recipient or sender of the communication.

**direct I/O** Block I/O that bypasses operating-system block features such as buffering and locking.

**direct memory access (DMA)** A resource-conserving and performance-improving operation for device controllers allowing devices to transfer large amounts of data directly to and from main memory.

**direct virtual memory access (DVMA)** DMA that uses virtual addresses rather than physical memory addresses as transfer sources and destinations.

**dirty bit** An MMU bit used to indicate that a frame has been modified (and therefore must have its contents saved before page replacement).

**discretionary access control (DAC)** Optional, as opposed to mandatory, access control.

**disinfecting** In the context of computer viruses, removing the components of a virus.

**disk arm** An HDD component that holds the read-write head and moves over cylinders of platters.

**disk image** Generally, the contents of a disk encapsulated in a file. In virtualization, a guest virtual machine's boot file system contents contained in a disk image.

**diskless** A term describing systems that have no local storage.

**dispatch latency** The amount of time the dispatcher takes to stop one thread and put another thread onto the CPU.

**dispatch queue** An Apple OS feature for parallelizing code; blocks are placed in the queue by Grand Central Dispatcher (GCD) and removed to be run by an available thread.

**dispatched** Selected by the process scheduler to be executed next.

**dispatcher** The kernel routine that gives control of a core to the thread selected by the scheduler.

**dispatcher objects** A Windows scheduler feature that controls dispatching and synchronization. Threads synchronize according to several different mechanisms, including mutex locks, semaphores, events, and timers.

**distributed denial-of-service attack (DDoS)** An attack from multiple sources (frequently a botnet of zombies) with the purpose of denying legitimate use of the attacked resource.

**distributed file system (DFS)** A file system that works across a network in which remote directories are visible from a local machine.

**distributed information system** A set of protocols providing unified information needed for remote computing.

**distributed lock manager (DLM)** A function used by a clustered system to supply access control and locking to ensure that no conflicting operations occur.

**distributed naming service** A set of protocols providing unified information needed for remote computing.

**distributed system** A collection of loosely coupled nodes interconnected by a communication network.

**distribution** A release of a version of an operating system.

**docker** An orchestration tool for containers.

**domain switching** The mechanism for switching dynamic domains.

**domain-name system (DNS)** The Internet system for resolving host names to host-ids, in which a distributed information system provides host-name-to-network-address translations for the Internet.

**double buffering** The copying of data twice (e.g., from a device to the kernel and then from the kernel to a process's address space), or the use of two buffers to decouple producers and consumers.

**double caching** The problem in which the same data might be in two different caches; solved by a unified buffer cache.

**double indirect block** In UFS, a block containing pointers to a single indirect block, which points to data blocks.

**down time** Generally, time during which a facility, system, or computing component are off-line and unavailable.

**driver end** The interface between STREAMS and the device being controlled.

**driver object** In Windows, the object representing a device driver.

**driver-registration system** In Linux, the kernel module facility that tells the rest of the kernel that a new driver is available.

**DTrace** A facility originally developed at Sun Microsystems that dynamically adds probes to a running system, both in user processes and in the kernel, for state analysis, performance tuning, and debugging.

**dual-booted** A term describing a computer that can boot one of two or more installed operating systems.

**dynamic loading** The loading of a process routine when it is called rather than when the process is started.

**dynamic random-access memory (DRAM)** The common version of RAM, which features high read and write speeds.

**dynamic storage-allocation** The class of file-block allocation methods that satisfy a request for n blocks from a list of free holes available.

**dynamic storage-allocation problem** The problem of how to satisfy a request for size n of memory from a list of free holes.

**dynamically linked libraries (DLLs)** System libraries that are linked to user programs when the processes are run, with linking postponed until execution time.

**earliest-deadline-first (EDF)** A real-time scheduling algorithm in which the scheduler dynamically assigns priorities according to completion deadlines.

**ease of use** The amount of difficulty and complexity involved in using some aspect of computing.

**EEPROM** Electrically erasable programmable read-only memory; a type of firmware.

**effective access time** The measured or statistically calculated time it takes to access something; e.g., see effective memory-access time.

**effective memory-access time** The statistical or real measure of how long it takes the CPU to read or write to memory.

**effective transfer rate** The actual, measured transfer rate of data between two devices (such as a computer and a disk drive).

**effective UID** The UID the process is currently using, which can be different from the login UID due to, e.g., escalating privileges.

**electrical storage** Storage devices made from electrical components, such as flash memory; one form of nonvolatile storage.

**ELF** The UNIX standard format for relocatable and executable files.

**embedded computer** A computer system within some other, larger system (such as a car) that performs specific, limited functions and has little or no user interface.

**emulation** A methodology used to enable a process to run when the compiled program's original (source) CPU type is different from the target CPU type on which the program is to run.

**emulator** A program that allows applications written for one hardware environment to run in a different hardware environment, such as a different type of CPU.

**encapsulate** In general, to enclose. In Java, encapsulation gives a class the ability to protect its data and methods from other classes loaded in the same JVM.

**encryption** The use of cryptography to limit the receivers of a message or access to data.

**entry section** The section of code within a process that requests permission to enter its critical section.

**entry set** In Java, the set of threads waiting to enter a monitor.

**environment vector** In Linux and UNIX, a list containing all of the environment variables of the shell that invoked a process (and are available to the process).

**environmental subsystem** In Windows, an operating environment that emulates a specific operating system by translating process API calls from that operating system to Windows calls.

**equal allocation** An allocation algorithm that assigns equal amounts of a resource to all requestors. In virtual memory, assigning an equal number of frames to each process.

**error-correcting code (ECC)** A value calculated from bytes of data and recalculated later to check for changes in the data.

**eSATA** A type of I/O bus.

**escalate privileges** To gain extra permissions for an activity, as when a user needs access to a device that is restricted.

**escape** Generally, a method of passing arbitrary commands or information when an interface does not provide a standard method.

**event latency** The amount of time between when an event occurs and when it is serviced.

**event** A Windows OS scheduling feature that is similar to a condition variable.

**eVM** An example of a partitioning hypervisor.

**exception** A software-generated interrupt caused either by an error (such as division by zero or invalid memory access) or by a specific request from a user program than an operating-system service be performed.

**exception dispatcher** The Windows component that processes exceptions.

**exclusive lock** A file lock similar to a writer lock in that only one process at a time can obtain the lock.

**executable and linkable format (ELF)** The UNIX standard format for relocatable and executable files.

**executable file** A file containing a program that is ready to be loaded into memory and executed.

**execution tracing** A Windows method of monitoring process execution and recording details for analysis and debugging.

**exit section** The section of code within a process that cleanly exits the critical section.

**expansion bus** A computer bus for connecting slow devices like keyboards.

**exponential average** A calculation used in scheduling to estimate the next CPU burst time based on the previous burst times (with exponential decay on older values).

**export list** The list of file systems available for remote mounting from a server.

**ext3** One of the most common types of Linux file systems.

**extended file attributes** Extended metadata about a file, including items such as character encoding details, file checksums, etc.

**extended file system** The most common class of Linux file systems, with ext3 and ext4 being the most commonly used file system types.

**extended file system (extfs)** The most common class of Linux file systems, with ext3 and ext4 being the most commonly used file system types.

**extensibility** The ability of an operating system to accommodate advances in computing technology via extensions (such as new kernel modules and new device drivers).

**extent** A chunk of contiguous storage space added to previously allocated space; a pointer is used to link the spaces.

**external data representation** A system used to resolve differences when data are exchanged between big- and little-endian systems.

**external fragmentation** Fragmentation in which available memory contains holes that together have enough free space to satisfy a request but no single hole is large enough to satisfy the request. More generally, the fragmentation of an address space into small, less usable chunks.

**fair scheduling** In the Completely Fair Scheduler, the scheduler algorithm that allots a proportion of the processor's time rather than time slices.

**false negatives** Results indicating that something is not a match to what is being detected, even though it is.

**false positives** Results indicating that something is a match to what is being detected, even though it isn't.

**fast directory sizing** In APFS, a feature that tracks and rapidly reports on directory sizes.

**fast I/O mechanism** In Windows, a high-speed bypass of the standard I/O handling mechanism in which the driver stack is called directly rather than having an IRP sent and processed.

**fault-tolerant system** A system that can suffer a failure of any single component and still continue operation.

**fiber** User-mode code that can be scheduled according to a user-defined scheduling algorithm.

**fibre channel (FC)** A type of storage I/O bus used in data centers to connect computers to storage arrays. A storage-attachment network.

**file** The smallest logical storage unit; a collection of related information defined by its creator.

**file attributes** Metadata about a file, such as who created it and when, file size, etc.

**file descriptor (fd)** UNIX open-file pointer, created and returned to a process when it opens a file.

**file handle** Windows name for the open-file file descriptor.

**file info window** A GUI view of the file metadata.

**file manipulation** A category of system calls.

**file mapping** In Windows, the first step in memory-mapping a file.

**file migration** A file system feature in which a file's location is changed automatically by the system.

**file object** The VFS representation of an open file.

**file reference** In Windows NTFS, a unique file ID that is the index of the file in the master file table (much like a UNIX inode number).

**file session** The series of accesses between open() and close() system calls that make up all the operations on a file by a process.

**file system** The system used to control data storage and retrieval; provides efficient and convenient access to storage devices by allowing data to be stored, located, and retrieved easily.

**File System Hierarchy Standard** A standard document specifying the overall layout of the standard Linux file systems.

**file-allocation table (FAT)** A simple but effective method of disk-space allocation used by MS-DOS. A section of storage at the beginning of each volume is set aside to contain the table, which has one entry per block, is indexed by block number, and contains the number of the next block in the file.

**file-control block (FCB)** A per-file block that contains all the metadata about a file, such as its access permissions, access dates, and block locations.

**file-organization module** A logical layer of the operating system responsible for files and for translation of logical blocks to physical blocks.

**file-server system** A server that provides a file-system interface where clients can create, update, read, and delete files (e.g., a web server that delivers files to clients running web browsers).

**filter drivers** In Windows, drivers allowed to insert themselves into the I/O processing chain.

**firewall** A computer, appliance, process, or network router that sits between trusted and untrusted systems or devices. It protects a network from security breaches by managing and blocking certain types of communications.

**firewall chains** In Linux, an ordered list of rules that specifies one of a number of possible firewall-decision functions plus matching components. A configuration of the firewall rules.

**firmware** Software stored in ROM or EEPROM for booting the system and managing low level hardware.

**first-come first-served (FCFS)** The simplest scheduling algorithm. The thread that requests a core first is allocated the core first.

**first-fit** In memory allocation, selecting the first hole large enough to satisfy a memory request.

**first-level interrupt handler** In some operating systems, an interrupt handler responsible for reception and queuing of interrupts; the interrupts are actually handled at another level (by the second-level handler).

**flash translation layer (FTL)** For nonvolatile memory, a table that tracks currently valid blocks.

**flow control** Generally, a method to pause a sender of I/O. In networking, a technique to limit the rate of data flow (e.g., to avoid buffer overflow and packet loss on a router).

**flush** Erasure of entries in, e.g., a TLB or other cache to remove invalid data.

**folder** A file system component that allows users to group files together.

**folder redirection** In Windows, for roaming users, a method for automatically storing a user's documents and other files on a remote server.

**foreground** Describes a process or thread that is interactive (has input directed to it), such as a window currently selected as active or a terminal window currently selected to receive input.

**foreground process** A process currently open and appearing on the display, with keyboard or other I/O device input directed to it.

**fork-join** A strategy for thread creation in which the main parent thread creates (forks) one or more child threads and then waits for the children to terminate and join with it.

**forward-mapped** Describes a scheme for hierarchical page tables in which address translation starts at the outer page table and moves inward.

**fourth extended file system (ext4)** In Linux, a current version of the extended file system, the successor to ext3.

**fragments** In networking, parts of messages. A message is split into fragments if it is too large to fit in one packet.

**frame table** In paged memory, the table containing frame details, including which frames are allocated, which are free, total frames in the system, etc.

**frame-allocation algorithm** The operating-system algorithm for allocating frames among all demands for frames.

**frames** Fixed-sized blocks of physical memory.

**free operating system** An operating system released under a license that makes its source code available and allows no-cost use, redistribution, and modification.

**free-behind** Sequential I/O performance optimization that removes a page or block from a buffer as soon as I/O to the next page is requested.

**free-frame list** A kernel-maintained data structure containing the list of currently available free frames of physical memory.

**free-space list** In file-system block allocation, the data structure tracking all free blocks in the file system.

**front-end processors** Small computers that perform certain tasks in an overall system; used by some systems to manage I/O and offload the general-purpose CPU.

**fsgid** In Linux, an added process property that allows file system access on behalf of another group.

**fsuid** In Linux, an added process property that allows file system access on behalf of another user.

**full backup** In file systems, a backup that includes all of the contents of a file system.

**functional language** A programming language that does not require states to be managed by programs written in that language (e.g., Erlang and Scala).

**fuzzing** A technique that tests for proper input validation (e.g., to avoid undetected buffer overruns).

**Galois field math** An advanced error-correcting calculation done in some forms of RAID.

**Gantt chart** A bar chart that can be used to illustrate a schedule.

**garbage collection** In general, recovery of space containing no-longer-valid data.

**general tree** A tree data structure in which a parent may have unlimited children.

**gestures** A user interface component in which motions cause computer actions (e.g., "pinching" the screen).

**gigabyte (GB)** 1,024^3 bytes.

**git** A version control system used for GNU/Linux and other programs.

**global replacement** In virtual memory frame allocation, allowing a process to select a replacement frame from the set of all frames in the system, not just those allocated to the process.

**GNU C compiler (gcc)** The standard C compiler of the GNU project, used throughout the industry on Linux and many other system.

**GNU General Public License (GPL)** A license agreement that codifies copylefting (allowing and requiring open sourcing of the associated programs); a common license under which free software is released.

**GNU/Linux (aka Linux)** An open-source operating system composed of components contributed by the GNU foundation and Linus Torvalds, as well as many others.

**Google Android** The mobile operating system created by Google Inc.

**Google file system (GFS)** A cluster-based distributed file system designed and used by Google.

**GPFS** A common commercial clustered file system by IBM.

**graceful degradation** The ability of a system to continue providing service proportional to the level of surviving hardware.

**graphical user interface (GUI)** A computer interface comprising a window system with a pointing device to direct I/O, choose from menus, and make selections and, usually, a keyboard to enter text.

**graphics processing unit (GPU)** A hardware component, sometimes part of the CPU and sometimes a separate device, that provides graphics computation and sometimes graphics output to a display.

**graphics shaders** Processes that provide the shading within graphics images.

**group** In file permissions, a collection of users that have access to a file based on assigned access rights.

**group identifier** Similar to a user identifier, but used to identify a group of users to determine access rights.

**group rights** In file permissions, the access rights belonging to a user group.

**GRUB** A common open-source bootstrap loader that allows selection of boot partitions and options to be passed to the selected kernel.

**guard pages** In Windows, no-access-allowed pages at the tops of the kernel-mode and user-mode stacks that detect stack overflows.

**guest** In virtualization, an operating system running in a virtual environment (rather than natively on the computer hardware).

**hacker** Someone attempting to breach computer security.

**Hadoop distributed file system (HDFS)** A cluster-based distributed file system used for big-data projects.

**Hadoop file system** An example of object storage management software.

**haiku** A three-line poem in which the first line contains five syllables, the second line contains seven syllables, and the third line contains five syllables.

**handle** Generally, an opaque value that provides access to an object or data structure; e.g., a file handle is returned when a file is opened and is a pointer to an entry in an open-file table.

**handle table** In Windows, a per-process handle table containing entries that track (by their handles) the objects used by the process.

**hard affinity** The situation in which an operating system allows a process's threads to run on the same processor at all times (as opposed to being moved to various processors).

**hard disk drive (HDD)** A secondary storage device based on mechanical components, including spinning magnetic media platters and moving read-write heads.

**hard error** An unrecoverable error (possibly resulting in data loss).

**hard links** File-system links in which a file has two or more names pointing to the same inode.

**hard real-time systems** Systems in which a thread must be serviced by its deadline; service after the deadline has expired is the same as no service at all.

**hard working-set limit** In Windows memory management, the maximum amount of physical memory that a process is allowed to use.

**hardware** The CPU, memory devices, input/output (I/O) devices, and any other physical components that are part of a computer.

**hardware objects** The CPU, memory devices, input/output (I/O) devices, and any other physical components that are part of a computer

**hardware threads** Threads associated with the processing core. A given CPU core may run multiple hardware threads to optimize core use—e.g., to avoid memory stalls by switching hardware threads if the current thread causes a stall.

**hardware transactional memory (HTM)** A transactional memory implementation using hardware cache hierarchies and cache coherency protocols to manage and resolve conflicts involving shared data residing in separate processors' caches.

**hardware-abstraction layer (HAL)** A kernel layer that isolates chipset-specific hardware aspects from general-purpose code.

**hash function** A function that takes data as its input, performs a numeric operation on the data, and returns a numeric value. Also, an algorithm for creating a hash (a small, fixed-size block of data calculated from a larger data set, used to determine if a message has been changed).

**hash map** A data structure that maps [key:value] pairs using a hash function; a hash function can then be applied to a key to obtain its matching value.

**hash value** The calculation resulting from a hash function.

**hashed page table** A page table that is hashed for faster access; the hash value is the virtual page number.

**head crash** On an HDD, a mechanical problem involving the read-write head touching a platter.

**heap section** The section of process memory that is dynamically allocated during process run time; it stores temporary variables.

**heartbeat** In computer clustering, a repeating signal to determine the state of the members of the cluster (e.g., to determine if a system is down).

**heterogeneous multiprocessing (HMP)** A feature of some mobile computing CPUs in which cores vary in their clock speeds and power management.

**high-availability** Describes a service that will continue even if one or more systems in the cluster fail.

**high-performance computing** A computing facility designed for use with a large number of resources to be used by software designed for parallel operation.

**high-performance event timer** A hardware timer provided by some CPUs.

**hit ratio** The percentage of times a cache provides a valid lookup (used, e.g., as a measure of a TLB's effectiveness).

**hives** In Windows, an internal repository of data.

**hole** In variable partition memory allocation, a contiguous section of unused memory. Also an alternative rock band formed by Courtney Love.

**honeypot** A false resource exposed to attackers; the resource appears real and enables the system to monitor and gain information about the attack.

**horizontal scalability** The ability to scale capacity not by expanding one item but by adding more items.

**host** In virtualization, the location of the virtual machine manager, which runs guest operating systems; generally, a computer.

**host bus adapter (HBA)** A device controller installed in a host bus port to allow connection of one or more devices to the host.

**host controller** The I/O-managing processors within a computer (e.g., inside a host bus adapter).

**host name** A human-readable name for a computer.

**host-attached storage** Storage accessed through local I/O ports (directly attached to a computer, rather than across a network or SAN).

**host-id** In networking, the unique number identifying a system on a network.

**hostname** The alphanumeric name for a system (such as becca.colby.edu).

**hot spare** An unused storage device ready to be used to recover data (e.g., in a RAID set).

**hot-standby mode** A condition in which a computer in a cluster does nothing but monitor the active server. If that server fails, the hot-standby host becomes the active server.

**huge pages** A feature that designates a region of physical memory where especially large pages can be used.

**hybrid cloud** A type of cloud computing that includes both public and private cloud components.

**hypercall** In paravirtualization, a call from a guest to the hypervisor to request a virtualization service, such as a page table change.

**hyper-threading** Intel's technology for assigning multiple hardware threads to a single processing core.

**hypervisor** The computer function that manages the virtual machine; also called a virtual machine manager (VMM).

**I/O burst** Scheduling process state in which the CPU performs I/O.

**I/O bus** A physical connection of an I/O device to a computer system.

**I/O channel** A dedicated, special-purpose CPU found in large systems like mainframes for performing I/O or offloading the general-purpose CPU.

**I/O control** A logical layer of the operating system responsible for controlling I/O, consisting of device drivers and interrupt handlers.

**I/O manager** In Windows, the system component responsible for I/O.

**I/O port** A hardware connector allowing connection of an I/O device.

**I/O request packet (IRP)** In Windows, a data structure to request file I/O that is sent from the I/O manager to the appropriate device driver.

**I/O subsystem** The I/O devices and the part of the kernel that manages I/O.

**I/O-bound process** A process that spends more of its time doing I/O than doing computations

**icons** Images representing objects (such as files or applications) that users can chose via the GUI.

**idempotent** Describes a function that, when applied more than once, has the same result every time.

**identifier** Generally, a numeric tag for a device or object. In networking, the unique host-id number identifying a system on a network.

**idle process** In Windows, a process that serves as the container of all idle threads.

**idle thread** In some operating systems, a special thread that runs on the CPU when no other thread is ready to run.

**immutable shared file** In a remote file system, a file that, once shared by its creator, cannot be modified.

**imperative language** Language for implementing algorithms that are state-based (e.g., C, C++, Java, and C#).

**impersonation** In Windows, the representation of a thread by a token for security purposes.

**implicit threading** A programming model that transfers the creation and management of threading from application developers to compilers and run-time libraries.

**incremental backup** In file systems, a backup that contains only some parts of a file system (the parts that have changed since the last full and incremental backups).

**indefinite blocking** A situation in which one or more processes or threads waits indefinitely within a semaphore.

**index** In file systems, an access method built on top of direct access in which a file contains an index with pointers to the contents of the file.

**index block** In indexed allocation, a block that contains pointers to the blocks containing the file's data.

**index root** In NTFS, the part of the directory containing the top level of the B+ tree.

**indexed allocation** In file-system block allocation, combining all block pointers in one or more index blocks to address limits in linked allocation and allow direct access to each block.

**indirect block** In UFS, a block containing pointers to direct blocks, which point to data blocks.

**InfiniBand (IB)** A high-speed network communications link.

**infinite blocking** A scheduling risk in which a thread that is ready to run is never put onto the

CPU due to the scheduling algorithm; it is starved for CPU time.

**information maintenance** A category of system calls.

**infrastructure as a service (IaaS)** A type of computing in which servers or storage are available over the Internet (e.g, storage available for making backup copies of production data).

**inode** In many file systems, a per-file data structure holding most of the metadata of the file. The FCB in most UNIX file systems.

**inode object** The VFS representation of an individual file.

**input/output operations per second** A measure of random access I/O performance; the number of inputs + outputs per second.

**integrity label** In Windows Vista and later versions, a mandatory access control component assigned to each securable object and subject.

**integrity levels** A mechanism, introduced in Windows Vista, that acts as a rudimentary capability system for controlling access.

**Intel 64** Intel 64 bit CPUs, part of a class of CPUs collectively known as x86-64

**interactive** Describes a type of computing that provides direct communication between the user and the system.

**intermachine interface** In distributed computing, a set of low-level functions for cross-machine interaction.

**internal fragmentation** Fragmentation that is internal to a partition.

**Internet** A worldwide system of interconnected computer networks.

**Internet key exchange (IKE)** A protocol that uses public key encryption to allow secure symmetric key exchange for IPSec.

**Internet protocol (IP)** The low-level networking protocol on which TCP and UDP are layered; responsible for routing IP datagrams through networks such as the Internet.

**Internet protocol security (IPSec)** A network protocol suite providing authentication and symmetric-key encryption of packets of network data.

**Internet Service Providers (ISPs)** Companies that provide Internet access.

**interpretation** A methodology that allows a program in computer language to be either executed in its high-level form or translated to an intermediate form rather than being compiled to native code.

**interprocess communication (IPC)** Communication between processes.

**interrupt** A hardware mechanism that enables a device to notify the CPU that it needs attention.

**interrupt address** A variable provided along with an interrupt signal to indicate the source of the interrupt.

**interrupt chaining** A mechanism by which each element in an interrupt vector points to the head of a list of interrupt handlers, which are called individually until one is found to service the interrupt request.

**Interrupt latency** The period of time from the arrival of an interrupt at the CPU to the start of the routine that services the interrupt.

**interrupt object** The Windows representation of an interrupt.

**interrupt priority level** Prioritization of interrupts to indicate handling order.

**interrupt request level (IRQL)** A prioritization method used in interrupt management.

**interrupt service routine (ISR)** An operating system routine that is called when an interrupt signal is received.

**interrupt vector** An operating-system data structure indexed by interrupt address and pointing to the interrupt handlers. A kernel memory data structure that holds the addresses of the interrupt service routines for the various devices.

**interrupt-controller hardware** Computer hardware components for interrupt management.

**interrupt-dispatch table** The Windows term for its interrupt vector.

**interrupt-handler routine** An operating system routine that is called when an interrupt signal is received.

**interrupt-request line** The hardware connection to the CPU on which interrupts are signaled.

**intruder** Someone attempting to breach security.

**intrusion prevention** The attempt to detect attempted and successful intrusions and properly respond to them.

**intrusion-prevention systems (IPS)** Systems to detect and prevent intrusions, usually in the form of self-modifying firewalls.

**inverted page table** A page-table scheme that has one entry for each real physical page frame in memory; the entry maps to a logical page (virtual address) value.

**iSCSI** The protocol used to communicate with SCSI devices; used across a network for more distant access.

**Itanium** Intel IA-64 CPU.

**iteration space** In Intel threading building blocks, the range of elements that will be iterated.

**Java virtual machine** In programming-environment virtualization, the process that implements the Java language and allows execution of Java code.

**job objects** In Windows, data structures for tracking collections of processes (e.g., to set CPU usage limits).

**job pool** The location where jobs are kept on disk while waiting for main memory to become available.

**job scheduling** The task of choosing which jobs to load into memory and execute.

**job** A set of commands or processes executed by a batch system.

**journaling** In a file system that features a write transaction log, logging of write activities for replay across actual file-system structures and consistency protection.

**journaling file system** A file system that features a write transaction log where all write activities are logged for replay across actual file-system structures and consistency protection.

**just-in-time (JIT)** In Java virtual machine implementations, describes a compiler that converts bytecode to native CPU instructions the first time the bytecode is interpreted to speed later execution.

**Kerberos** A network authentication protocol invented at M.I.T. that forms the basis for the Microsoft network authentication protocol.

**kernel** The operating system component running on the computer at all times after system boot.

**kernel abstractions** Components provided with the Mach microkernel to add functionality beyond the microkernel, such as tasks, threads, memory objects, and ports.

**kernel environment** In the layered macOS and iOS operating system design, the Darwin layer that includes the Mach microkernel and the BSD UNIX kernel.

**kernel extensions (kexts)** Third-party components added to the kernel (usually to support third-party devices or services).

**kernel mode** A CPU mode in which all instructions are enabled. The kernel runs in this mode. See also user mode.

**kernel threads** Threads running in kernel mode.

**kernel-mode driver framework (KMDF)** A framework in Windows to facilitate the writing of kernel-mode device drivers.

**kernel-mode thread (KT)** In Windows, the name for the state of a thread when it is running in kernel mode.

**Kernighan's Law** "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

**keys** In the context of protection, unique bit patterns held by domains corresponding with unique bit patterns (locks) held by objects. Generally, secrets used in cryptography.

**keystream** An infinite set of bits used to encrypt a plain-text stream through an XOR operation in a stream cipher.

**keystroke logger** A program that captures keystrokes entered by users.

**kilobyte (KB)** 1,024 bytes.

**Kubernetes** An orchestration tool for containers.

**labels** In mandatory access control, identifiers assigned to objects and/or subjects. The label is checked by the operating system when an operation is requested to determine if it is allowed.

**layered approach** A kernel architecture in which the operating system is separated into a number of layers (levels); typically, the bottom layer (layer 0) is the hardware, and the highest (layer N) is the user interface.

**lazy swapper** A swapping method in which only pages that are requested are brought from secondary storage into main memory.

**least frequently used (LFU)** In general, an algorithm that selects the item that has been used least frequently. In virtual memory, when access counts are available, selecting the page with the lowest count.

**least privilege** Design principle stating that every program and every privileged user of the system should operate using the least amount of privilege necessary to complete the job.

**least recently used (LRU)** In general, an algorithm that selects the item that has been used least recently. In memory management, selecting the page that has not been accessed in the longest time.

**lgroups** In Solaris, locality groups located in the kernel; each lgroup gathers together CPUs and memory, and each CPU in that group can access any memory in the group within a defined latency interval. A method for dealing with NUMA.

**library operating systems** The applications that run on unikernels, containing both the kernel and the application code.

**lightweight directory-access protocol (LDAP)** A secure distributed naming service used throughout the computer industry.

**lightweight process (LWP)** A virtual processor-like data structure allowing a user thread to map to a kernel thread.

**limit register** A CPU register that defines the size of the range. Together with the base register, it defines the logical address space.

**line discipline** In Linux, an interpreter for the information from a terminal device.

**link** In file naming, a file that has no contents but rather points to another file.

**linked allocation** A type of file-system block allocation in which each file is a linked list of allocated blocks containing the file's contents. Blocks may be scattered all over the storage device.

**linked list** A data structure in which items are linked to one another.

**linker** A system service that combines relocatable object files into a single binary executable file.

**Linux distribution** A Linux system plus administrative tools to simplify the installation, upgrading, and management of the system.

**Linux instance** A set of Pico processes running in Windows created by WSL containing an init process and a bash shell (allowing executing of Linux commands within Windows)

**Linux kernel** The operating-system kernel of a Linux system.

**Linux system** The kernel, programs, and files that comprise a complete, runnable Linux system.

**list** A data structure that presents a collection of data values as a sequence.

**little-endian** A system architecture that stores the least significant byte first in a sequence of bytes.

**Little's formula** A scheduling equation ($n = ? \times W$) that is particularly useful because it is valid for any scheduling algorithm and arrival distribution.

**live migration** In virtualization, the movement of a running guest between two separate physical hosts.

**LiveCD** An operating system that can be booted and run from a CD-ROM (or more generally from any media) without being installed on a system's boot disk(s).

**LiveDVD** An operating system that can be booted and run from a DVD (or more generally from any media) without being installed on a system's boot disk(s).

**livelock** A condition in which a thread continuously attempts an action that fails.

**liveness** A set of properties that a system must satisfy to ensure that processes make progress during their execution life cycle.

**living document** A document that is modified over time to keep it up to date.

**load balancing** The movement of jobs or network packets between various components (say, computers in a network) to distribute the load or route around failures. Load balancing attempts to keep the workload evenly distributed across all processors in an SMP system.

**load sharing** The ability of a system with multiple CPU cores to schedule threads on those cores.

**loadable kernel module (LKM)** A kernel structure in which the kernel has a set of core components and can link in additional services via modules, either at boot time or during run time.

**loader** A system service that loads a binary executable file into memory, where it is eligible to run on a CPU core.

**local procedure call** In Windows, a method used for communication between two processes on the same machine.

**local replacement** In virtual-memory frame allocation, allowing a process to select a replacement frame only from the set of all frames allocated to the process.

**local replacement algorithm** A virtual-memory page replacement algorithm that avoids thrashing by not allowing a process to steal frames from other processes.

**local-area network (LAN)** A network that connects computers within a room, a building, or a campus.

**locality** The tendency of processes to reference memory in patterns rather than randomly.

**locality model** A model for page replacement based on the working-set strategy.

**locality of reference** The tendency of processes to reference memory in patterns rather than randomly.

**location independence** In distributed computing, a feature in which the name of an object does not need to be changed when the object's physical location changes.

**location transparency** In distributed computing, a feature in which the name of an object does not reveal its physical location.

**lock** A mechanism that restricts access by processes or subroutines to ensure integrity of shared data.

**locked** In general, fixed in place. In memory management, pages can be locked into memory to prevent them from being paged out.

**lock-free** An algorithm that provides protection from race conditions without requiring the overhead of locking.

**locking** Protecting critical sections, data structures, or objects though the use of code that coordinates access.

**lock-key scheme** In protection, a compromise between access lists and capability lists in which each object has a unique bit pattern (a lock) and each domain has a unique bit pattern (a key).

**log file** A file containing error or "logging" information; used for debugging or understanding the state or activities of the system.

**log-based transaction-oriented file system** A file system that features a write transaction log where all write activities are logged for replay across actual file-system structures and consistency protection.

**logic bomb** A remote-access tool designed to operate only when a specific set of logical conditions is met.

**logical address** Address generated by the CPU; must be translated to a physical address before it is used.

**logical address space** The set of all logical addresses generated by a program.

**logical blocks** Logical addresses used to access blocks on storage devices.

**logical cluster numbers** In Windows, the name given to secondary storage physical addresses.

**logical file system** A logical layer of the operating system responsible for file and file-system metadata management; maintains the FCBs.

**logical formatting** The creation of a file system in a volume to ready it for use.

**logical memory** Memory as viewed by the user; usually a large uniform array, not matching physical memory in virtual memory systems.

**logical records** File contents logically designated as fixed-length structured data.

**LOOK** An HDD I/O scheduling algorithm modification of SCAN that stops the head after the last request is complete (rather than at the innermost or outermost cylinder).

**loopback** Communication in which a connection is established back to the sender.

**loosely coupled** Describes a kernel design in which the kernel is composed of components that have specific and limited functions.

**lottery scheduling** A scheduling algorithm in which "lottery tickets" are given to threads and a lottery number is chosen at random to determine the next thread to get CPU time.

**low-fragmentation heap (LFH)** An optimization of the Windows default heap designed to decrease fragmentation.

**low-level formatting** The initialization of a storage medium in preparation for its use as a computer storage device.

**Lustre** A common open-source clustered file system.

**LXC** A Linux container technology.

**Mach** An operating system with microkernel structure and threading; developed at Carnegie Mellon University.

**Mach-O** The macOS format of executable files.

**magic cookie** A crude method of storing a text string at the start of a text file to indicate the type of text in the file.

**magic number** A crude method of storing a number at the start of a file to indicate the type of the data in the file.

**magnetic tape** A magnetic media storage device consisting of magnetic tape spooled on reels and passing over a read-write head. Used mostly for backups.

**main queue** Apple OS per-process block queue.

**main TLB** ARM CPU outer-level TLB; checked after the micro TLB lookup and before a miss causes a page table walk.

**mainframe** The largest class of computers (along with supercomputers), hosting hundreds of users and many and/or large jobs.

**major fault** In virtual memory, a page fault that can be resolved without having to page in data from secondary storage.

**malware** Software designed to exploit, disable, or damage computer systems.

**mandatory access control (MAC)** Access control settings enforced in the form of system policy.

**mandatory file-lock mechanism** A file-locking system in which the operating system enforces locking and file access.

**man-in-the-middle attack** An attack in which the attacker sits in the middle of the data flow of a communication, masquerading as the sender to the receiver and vice versa.

**MapReduce** A Google-created big data programming model and implementation for parallel processing across nodes in a distributed cluster. A layer on top of the Google file system (GFS), it allows developers to carry out large-scale parallel computations easily.

**marshaling** Packaging a communication into an expected format for transmittal and reception.

**maskable** Describes an interrupt that can be delayed or blocked (such as when the kernel is in a critical section).

**masquerading** A practice in which a participant in a communication pretends to be someone else (another host or another person).

**master boot record (MBR)** Windows boot code, stored in the first sector of a boot partition.

**master file directory (MFD)** In two-level directory implementation, the index pointing to each UFD.

**master file table** The NTFS volume control block.

**master key** In the lock-key protection scheme, a key that is associated with each object and can be defined or replaced with the set-key operation to revoke or change access rights.

**matchmaker** A function that matches a caller to a service being called (e.g., a remote procedure call attempting to find a server daemon).

**mean time between failure (MTBF)** The statistical mean time that a device is expected to work correctly before failing.

**mean time to data loss** The statistical mean of the time until data is lost.

**mean time to repair** The statistical mean of the time to repair a device (e.g., to get a replacement and install it).

**mechanical storage device** A storage device based on moving mechanical parts (such as HDDs, optical disks, and magnetic tape); one form of nonvolatile storage.

**mechanism** An operation that defines how something will be done.

**medium access control (MAC) address** A unique byte number assigned to every Ethernet device allowing it to be located by packets sent across a LAN.

**megabyte (MB)** 1,024^2 bytes.

**memory** Volatile storage within a computer system.

**memory barriers** Computer instructions that force any changes in memory to be propagated to all other processors in the system.

**memory compression** In memory management, an alternative to paging involving compressing the contents of frames to decrease the memory used.

**memory compression process** In Windows 10, a process that maintains a working set of compressed standby pages.

**memory fences** Computer instructions that force any changes in memory to be propagated to all other processors in the system.

**memory manager (MM)** The Windows name for the component that manages memory.

**memory mapping** A file-access method in which a file is mapped into the process memory space so that standard memory access instructions read and write the contents of the file; an alternative to the use of read() and write() calls.

**memory model** Computer architecture memory guarantee, usually either strongly ordered or weakly ordered.

**memory resident** Objects, such as pages, that are in main memory and ready for threads to use or execute.

**memory stall** An event that occurs when a thread is on CPU and accesses memory content that is not in the CPU's cache. The thread's execution stalls while the memory content is fetched.

**memory transaction** A sequence of memory read–write operations that are atomic.

**memory-management unit (MMU)** The hardware component of a computer CPU or motherboard that allows it to access memory.

**memory-mapped file** A file that is loaded into physical memory via virtual memory methods, allowing access by reading and writing to the memory address occupied by the file.

**memory-mapped I/O** A device I/O method in which device-control registers are mapped into the address space of the processor.

**message** In networking, a communication, contained in one or more packets, that includes source and destination information to allow correct delivery. In message-passing communications, a packet of information with metadata about its sender and receiver.

**message digest** The calculation resulting from a hash function.

**message passing** In interprocess communication, a method of sharing data in which messages are sent and received by processes. Packets of information in predefined formats are moved between processes or between computers.

**message-authentication code (MAC)** A cryptographic checksum calculated in symmetric encryption; used to authenticate short values.

**message-passing model** A method of interprocess communication in which messages are exchanged.

**metadata** A set of attributes of an object. In file systems, e.g., all details of a file except the file's contents.

**metaslabs** Chunks of blocks. In ZFS, a pool of storage is split into metaslabs for easier management.

**methods** In Java, functions that act on objects and data fields.

**metropolitan-area network (MAN)** A network linking buildings within a city.

**micro TLB** ARM CPU inner-level TLBs, one for instructions and one for data.

**microkernel** An operating-system structure that removes all nonessential components from the kernel and implements them as system and user-level programs.

**Microsoft interface definition language** The Microsoft text-based interface definition language; used, e.g., to write client stub code and descriptors for RPC.

**middleware** A set of software frameworks that provide additional services to application developers.

**minicomputer** A mid-sized computer, smaller than a mainframe but larger (in resources and users) than a workstation.

**minidisks** Virtual disks used in early IBM virtual systems.

**minimum granularity** In the Completely Fair Scheduler, a configurable variable representing the minimum length of time any process is allocated to the processor.

**miniport driver** In Windows I/O, the device-specific driver.

**minor fault** In virtual memory, a page fault resolved by executing an I/O to bring in the page from secondary storage.

**mirrored volume** A volume in which two devices are mirrored.

**mirroring** In storage, a type of RAID protection in which two physical devices contain the same content. If one device fails, the content can be read from the other.

**mobile computing** A mode of computing involving small portable devices like smartphones and tablet computers.

**mode bit** A CPU status bit used to indicate the current mode: kernel (0) or user (1).

**modify bit** An MMU bit used to indicate that a frame has been modified (and therefore must have its contents saved before page replacement).

**module loader and unloader** In Linux, user-mode utilities that work with the module-management system to load modules into the kernel.

**module-management system** In Linux, the facility that allows kernel modules to be loaded into memory and to communicate with the rest of the kernel.

**monitor** A high-level language synchronization construct that protects variables from race conditions.

**monitor call** A software-triggered interrupt allowing a process to request a kernel service.

**monoculture** A community of computer systems that are very similar to one another. This similarity makes them easier to attack and thus represents a threat to security.

**monolithic** In kernel architecture, describes a kernel without structure (such as layers or modules).

**Moore's Law** A law predicting that the number of transistors on an integrated circuit would double every eighteen months.

**most frequently used (MFU)** In general, an algorithm that selects the item that has been used most frequently. In virtual memory, when access counts are available, selecting the page with the highest count.

**mount point** The location within the file structure where a file system is attached.

**mount protocol** The protocol for mounting a file system in a remote file system.

**mount table** An in-memory data structure containing information about each mounted volume. It tracks file systems and how they are accessed.

**mounting** Making a file system available for use by logically attaching it to the root file system.

**multicore** Multiple processing cores within the same CPU chip or within a single system.

**multicore processor** Multiple processing cores within the same CPU chip.

**multicore systems** Systems that have two or more hardware processors (CPU cores) in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices.

**multifactor authentication** Authentication based on two or more sources of data, with more sources generally providing stronger authentication.

**multilevel feedback queue** A scheduling algorithm that allows a process to move between queues.

**multilevel queue** A scheduling algorithm that partitions the ready queue into several separate queues.

**multiple universal-naming-convention provider (MUP)** The component within Windows that executes remote file accesses.

**multiple user interface (MUI)** A Windows Vista feature that allows multiple user interfaces, possibly configured for different locales, to be used concurrently.

**multiprocessor** Multiple processors within the same CPU chip or within a single system.

**multiprocessor systems** Systems that have two or more hardware processors (CPU cores) in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices.

**multiprogramming** A technique that increases CPU utilization by organizing jobs (code and data) so that the CPU always has a job to execute.

**multitasking** The concurrent performance of multiple jobs. A CPU executes multiple jobs by switching among them, but the switches occur so frequently that users can interact with the processes.

**multithreaded** A term describing a process or program with multiple threads of control, allowing multiple simultaneous execution points.

**mutex lock** A mutual exclusion lock; the simplest software tool for assuring mutual exclusion.

**mutual exclusion** A property according to which only one thread or process can be executing code at once.

**name server** In the domain-name system, a host or software that provides resolving services.

**named pipes** A connection-oriented messaging mechanism—e.g., allowing processes to communicate within a single computer system.

**named semaphore** A POSIX scheduling construct that exists in the file system and can be shared by unrelated processes.

**named shared-memory object** In Windows API, a section of a memory-mapped file accessible by name from multiple processes.

**namespace** In Linux, a process's specific view of the file system hierarchy.

**naming** In distributed computing, the mapping between logical and physical objects.

**national-language-support (NLS)** A Windows API providing support for localization (including date, time, and money formats).

**need-to-know principle** The principle that only those resources currently needed should be available to use at a given time.

**nested page tables (NPTs)** In virtualization, a method used by the virtual machine manager to maintain page-table state both for guests and for the system.

**network** In the simplest terms, a communication path between two or more systems.

**network address translation** In networking, the mapping of one header address to another by modifying the network packets (e.g., allowing a host to provide IP addresses to multiple guests while presenting only one IP address to the connected network).

**network computer** A limited computer that understands only web-based computing.

**network device interface specification (NDIS)** An internal Windows networking interface separating network adapters from transport protocols.

**network devices** I/O devices that connect to a network.

**network file system (NFS)** A common network file system used by UNIX, Linux, and other operating systems to share files across a network.

**network operating system** A type of operating system that provides features such as file sharing across a network, along with a communication scheme that allows different processes on different computers to exchange messages. It provides an environment in which users can access remote resources by remote login or data transfer between remote and local systems.

**network time protocol** A network protocol for synchronizing system clocks.

**network virtual memory** A distributed computing feature similar to virtual memory but with the backing store on a remote system.

**network-attached storage (NAS)** Storage accessed from a computer over a network.

**NFS protocol** The protocol used for remote file access, remote file system mounting, etc., by the NFS file system.

**nice value** One of a range of values from ?20 to +19, where a numerically lower value indicates a higher relative scheduling priority.

**NIS** A distributed naming service that provides username, password, hostname, and printer information to a set of computers.

**nonblocking** A type of I/O request that allows the initiating thread to continue while the I/O operation executes. In interprocess communication, a communication mode in which the sending process sends the message and resumes operation and the receiver process retrieves either a valid message or a null if no message is available. In I/O, a request that returns whatever data is currently available, even if it is less than requested.

**noncontainer objects** In Windows 10, a category of objects that cannot contain other objects.

**nonmaskable interrupt** An interrupt that cannot be delayed or blocked (such as an unrecoverable memory error)

**nonpreemptive** Scheduling in which, once a core has been allocated to a thread, the thread keeps the core until it releases the core either by terminating or by switching to the waiting state.

**nonpreemptive kernels** A type of kernel that does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

**nonrepudiation** Proof that an entity performed an action (frequently performed by digital signatures).

**non-uniform memory access (NUMA)** An architecture aspect of many computer systems in which the time to access memory varies based on which core the thread is running on (e.g., a core interlink is slower than accessing DIMMs directly attached to core).

**nonvolatile memory (NVM)** Persistent storage based on circuits and electric charges.

**nonvolatile storage (NVS)** Storage in which data will not be lost in a power outage or similar event.

**NOOP** The Linux NVM scheduling algorithm, first come first served but with adjacent requests merged into fewer, larger I/O requests.

**NUMA node** One or more cores (e.g., cores that share a cache) that are grouped together as a scheduling entity for affinity or other uses.

**NVM express (NVMe)** A high-speed I/O bus for NVM storage.

**NVRAM** DRAM with battery or other backup power, rendering it nonvolatile.

**object** An instance of a class or an instance of a data structure. In Windows and generally, an instance of an object type.

**object linking and embedding (OLE)** A Microsoft technology allowing services to provide functions to components (e.g., for inserting spreadsheets into Word documents).

**object manager** In Windows, the kernel (executive) component that manipulates objects.

**object type** In Windows, a system-defined data type that has a set of attributes and methods that help define its behavior.

**off-line** Generally, a facility, system, or computing component that is unavailable. In file system implementation, operations executing while the file system is unavailable for use.

**one-time password** A password that is only valid once.

**on-line** Generally, a facility, system, or computing component that is available. In file system implementation, operations executing while the file system is available for use.

**open count** The number of processes having an open file.

**open-file table** An operating system data structure containing details of every file open within the system.

**open-source operating system** An operating system or other program available in source-code format rather than as compiled binary code.

**operating system** A program that manages a computer’s hardware, provides a basis for application programs, and acts as an intermediary between the computer user and the computer hardware.

**optimal page-replacement algorithm** A theoretically optimal page replacement algorithm that has the lowest page-fault rate of all algorithms and never suffers from Belady’s anomaly.

**Orange Book** U.S. Department of Defense Trusted Computer System Evaluation Criteria; a method of classifying the security of a system design.

**orphan** The child of a parent process that terminates in a system that does not require a terminating parent to cause its children to be terminated.

**OS/2** A PC operating system from the mid 1980s co-developed by IBM and Microsoft to replace MS-DOS; generally considered to be a failure.

**OSI protocol stack** A set of cooperating network protocols that form a fully functional network. The OSI model formalized some of the earlier work done in network protocols but is currently not in widespread use.

**out-of-band** In networking, a term describing data delivered in a manner independent of the main data stream (e.g., delivery of a symmetric key in a paper document).

**out-of-memory (OOM) killer** In Linux, a routine that executes when free memory is very low, terminating processes to free memory.

**over-allocating** Generally, providing access to more resources than are physically available. In virtual memory, allocating more virtual memory than there is physical memory to back it.

**overcommitment** In virtualization, providing resources to guests that exceed available physical resources.

**over-provisioning** In non-volatile memory, space set aside for data writes that is not counted in the device free space.

**owner** In file permissions, the userid that owns and controls access to a file.

**owner rights** In file permissions, the userid that owns and controls access to a file.

**page address extension (PAE)** Intel IA-32 CPU architecture hardware that allows 32-bit processors to access physical address space larger than 4GB.

**page allocator** The kernel routine responsible for allocating frames of physical memory.

**page cache** In file I/O, a cache that uses virtual memory techniques to cache file data as pages rather than file-system-oriented blocks for efficiency.

**page directory** In Intel IA-32 CPU architecture, the outermost page table.

**page directory pointer table** PAE pointer to page tables.

**page fault** A fault resulting from a reference to a non-memory-resident page of memory.

**page frame** A Windows virtual memory data structure.

**page frame number (PFN)** In Windows, the name of the indicator of the page frame address.

**page number** Part of a memory address generated by the CPU in a system using paged memory; an index into the page table.

**page offset** Part of a memory address generated by the CPU in a system using paged memory; the offset of the location within the page of the word being addressed.

**page replacement** In virtual memory, the selection of a frame of physical memory to be replaced when a new page is allocated.

**page slot** In Linux swap-space management, a part of the data structure tracking swap-space use.

**page table** In paged memory, a table containing the base address of each frame of physical memory, indexed by the logical page number.

**page-directory entry (PDE)** A Windows virtual-memory data structure.

**page-fault frequency** The frequency of page faults.

**page-fault rate** A measure of how often a page fault occurs per memory access attempt.

**pageout policy** Generally, an algorithm for deciding which memory pages to page out. In Linux, the virtual memory pageout policy uses a modified version of the second-chance algorithm.

**pager** The operating-system component that handles paging.

**page-replacement algorithm** In memory management, the algorithm that chooses which victim frame of physical memory will be replaced by a needed new frame of data.

**page** A fixed-sized block of logical memory.

**page-table base register** In paged memory, the CPU register pointing to the in-memory page table.

**page-table entry (PTE)** A Windows virtual memory data structure.

**page-table length register** A CPU register indicating the size of the page table.

**paging** A common memory management scheme that avoids external fragmentation by splitting physical memory into fixed-sized frames and logical memory into blocks of the same size called pages.

**paging file** The Windows term for backing store.

**paging mechanism** In Linux, the kernel component that carries out the transfer of pages back and forth to backing store.

**paired password** In authentication, a challenge-response set of secret keys, where only the correct response to the challenge provides authentication.

**parallel file system (PFS)** A file system that is LAN-based and treats N systems storing data and Y clients accessing the data as a single client-server instance; more complex than a client-server DFS but less complex than a cluster-based DFS. GPFS and Lustre are examples.

**parallel regions** Blocks of code that may run in parallel.

**parallel systems** Systems that have two or more hardware processors (CPU cores) in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices.

**parallelization** The process of dividing a program into separate components that run in parallel on individual cores in a computer or computers in a cluster.

**paravirtualization** A technique in which a guest operating system is modified to work in cooperation with a virtual machine manager.

**parent** In a tree data structure, a node that has one or more nodes connected below it.

**partition** Logical segregation of storage space into multiple area; e.g., on HDDs, creating several groups of contiguous cylinders from the devices' full set of cylinders.

**partition boot sector** The NTFS boot control block.

**passphrase** A longer, generally more secure password composed of multiple words.

**password** A secret key, usually used to authenticate a user to a computer.

**path name** The file-system name for a file, which contains all the mount-point and directory-entry information needed to locate the file (e.g., "C:/foo/bar.txt" and "/foo/bar.txt")

**path-name translation** The parsing of a file name into separate directory entries, or components.

**PCIe bus** A common computer I/O bus connecting the CPU to I/O devices.

**peer-to-peer (p2p)** A mode of distributed computing in which all nodes act as both clients of other nodes and servers to other nodes.

**penetration test** The scanning of a target entity to look for known vulnerabilities.

**performance tuning** The activity of improving performance by removing bottlenecks.

**periodic** A type of real-time process that repeatedly moves at fixed intervals between two modes: needing CPU time and not needing CPU time.

**permissions** An entity's access rights to an object (e.g., a user's access rights to a file).

**per-process open-file table** A kernel in-memory per-process data structure containing pointers to the system-wide open-file table, as well as other information, for all files the process has open.

**personal firewall** A software layer, either part of the operating system or added to a computer, limiting communication to and from a given host.

**personal identification number** A usually short and not very secure password composed of some combination of digits 0-9.

**personal-area network (PAN)** A network linking devices within several feet of each other (e.g., on a person).

**petabyte (PB)** 1,024^5 bytes.

**Peterson's solution** A historically interesting algorithm for implementing critical sections.

**phishing** A class of social engineering attacks in which a legitimate-looking e-mail or website tricks a user into breaching confidentiality or enabling privilege escalation.

**PHY** The physical hardware component that connects to a network (implements layer 1 in the OSI model).

**physical address** Actual location in physical memory of code or data.

**physical address space** The set of all physical addresses generated by a program.

**physical formatting** The initialization of a storage medium in preparation for its use as a computer storage device.

**physical-to-virtual (P-to-V)** In virtualization, the conversion of a physical system's operating system and applications to a virtual machine.

**Pico** In WSL, a special Linux-enabling process that translates Linux system calls to the LXCore and LXSS services.

**pinning** In memory management, locking pages into memory to prevent them from being paged out.

**pipe** A logical conduit allowing two processes to communicate.

**platform as a service (PaaS)** A software stack ready for application use via the Internet (e.g., a database server).

**platter** An HDD component that has a magnetic media layer for holding charges.

**plug-and-play (PnP) manager** In Windows, the manager responsible for detecting and enumerating devices when the system is booting and adding and removing devices when the system is running.

**pluggable authentication module (PAM)** A shared library that can be used by any system component to authenticate users.

**plug-in** An add-on functionality that expands the primary functionality of a process (e.g., a web browser plug-in that displays a type of content different from what the browser can natively handle).

**point-to-point tunneling protocol (PPTP)** A protocol in Windows and other systems allowing communication between remote-access server modules and client systems connected across a WAN.

**policy** A rule that defines what will be done.

**policy algorithm** In Linux, a part of the paging system that decides which pages to write out to backing store and when to write them.

**polling** An I/O loop in which an I/O thread continuously reads status information waiting for I/O to complete.

**pool** In virtual memory, a group of free pages kept available for rapid allocation (e.g., for copy-on-write). In ZFS, drives, partitions, or RAID sets that can contain one or more file systems.

**pop** The action of removing an item from a stack data structure.

**port** A communication address; a system may have one IP address for network connections but many ports, each for a separate communication. In computer I/O, a connection point for devices to attach to computers. In software development, to move code from its current platform to another platform (e.g., between operating systems or hardware systems). In the Mach OS, a mailbox for communication.

**port driver** In Windows I/O, the common driver for a class of devices.

**port number** In TCP/IP and UDP/IP networking, an address of a service on a system.

**port set** A collection of ports, as declared by a task, that can be grouped together and treated as one port for the purposes of the task.

**portable** An aspect of software that describes its ease of transfer between CPU architectures and computer systems.

**portable executable (PE)** The Windows format for executable files.

**portals** Gateways between requestors and services running on provider computers.

**position-independent code (PIC)** In Linux, binary code compiled from shared libraries that can be loaded anywhere in memory.

**positioning time** On an HDD, the time it takes the read-write head to position over the desired track.

**power manager** In Windows, the component that implements power management policies.

**power users** Users with unusually deep knowledge of a system.

**power-of-2 allocator** In the buddy system, an allocator that satisfies memory requests, in units sized as a power of 2, from a fixed-sized segment consisting of contiguous pages.

**power-on self-test (POST)** A firmware routine run at system power-on that tests the system for hardware issues, identifies and initializes many of the attached devices, and builds the description of the devices used by the advanced configuration and power interface (ACPI).

**preemptive** A form of scheduling in which processes or threads are involuntarily moved from the running state (e.g., by a timer signaling the kernel to allow the next thread to run).

**preemptive kernel** A type of kernel that allows a process to be preempted while it is running in kernel mode.

**preemptive multitasking** A model of multitasking in which threads on cores may be preempted by higher-priority threads before finishing their scheduled time quanta.

**prepaging** In virtual memory, bringing pages into memory before they are requested.

**principle of least privilege** A design principle stating that every program and every privileged user of the system should operate using the least amount of privilege necessary to complete the job.

**priority inversion** A scheduling challenge arising when a higher-priority process needs to read or modify kernel data that are currently being accessed by a lower-priority process.

**priority number** A number indicating the position of a process in a conditional-wait queue in a monitor construct.

**priority paging** Prioritizing selection of victim frames based on some criteria, such as avoiding selection of shared library pages.

**priority replacement algorithm** A virtual memory page replacement algorithm that avoids thrashing by not allowing a process to steal frames from other processes.

**priority-inheritance protocol** A protocol for solving priority inversion in which all processes that are accessing resources needed by a higher-priority process inherit that higher priority until they are finished with the resources in question.

**priority scheduling** A scheduling algorithm in which a priority is associated with each thread and the free CPU core is allocated to the thread with the highest priority.

**private cloud** Cloud computing run by a company for that company's own use.

**private key** In an asymmetric encryption algorithm, a key that must be kept private for use in authenticating, encrypting, and decrypting.

**privilege escalation** The enabling of more privileges than an entity (process, system, person) should have.

**privileged instructions** Instructions that can execute only if the CPU is in in kernel mode.

**privileged mode** A CPU mode in which all instructions are enabled. The kernel runs in this mode. See also user mode.

**proc file system (/proc)** A pseudo file system using file-system interfaces to provide access to a system's process name space.

**procedural language** A language that implements state-based algorithms (e.g., C, C++, Java, and C#).

**process** A program loaded into memory and executing.

**process control** A category of system calls.

**process control block** A per-process kernel data structure containing many pieces of information associated with the process.

**process identifier (pid)** A unique value for each process in the system that can be used as an index to access various attributes of a process within the kernel.

**process lifetime management (PLM)** A Windows power-saving feature that suspends all threads within a process that has not been used for a few seconds.

**process migration** The movement of a process between computers.

**process name** A human-readable name for a process.

**process scheduler** A scheduler that selects an available process (possibly from a set of several processes) for execution on a CPU.

**process synchronization** Coordination of access to data by two or more threads or processes.

**process-contention scope (PCS)** A scheduling scheme, used in systems implementing the many-to-one and many-to-many threading models, in which competition for the CPU takes place among threads belonging to the same process.

**processor affinity** A kernel scheduling method in which a process has an affinity for (prefers) the processor on which it is currently running.

**processor groups** In Windows 7, processors grouped together for management and scheduling.

**producer** A process role in which the process produces information that is consumed by a consumer process.

**production kernels** Kernels released for production use (as opposed to development use).

**profiling** Periodically sampling the instruction pointer to determine which code is being executed; used in debugging and performance tuning.

**program counter** A CPU register indicating the main memory location of the next instruction to load and execute.

**programmable interval timer** A hardware timer provided by many CPUs.

**programmed I/O (PIO)** A method of transferring data between a CPU and a peripheral device in which data are transferred one byte at a time.

**programming-environment virtualization** Virtualization in which a virtual machine manager does not virtualize real hardware but instead creates an optimized virtual system (examples include Oracle Java and Microsoft.Net).

**project** In Solaris scheduling, a group of processes scheduled as a unit.

**proportional allocation** An allocation algorithm that assigns a resource in proportion to some aspect of the requestor. In virtual memory, the assignment of page frames in proportion to the size each process.

**proportional share** A scheduler that operates by allocating T shares among all applications, ensuring that each gets a specific portion of CPU time.

**protection** A category of system calls. Any mechanism for controlling the access of processes or users to the resources defined by a computer system.

**protection domain** In protection, a set of resources that a process may access. In virtualization, a virtual machine manager creates a protection domain for each guest to inform the CPU of which physical memory pages belong to that guest.

**protection mask** In Linux and UNIX, a set of bits assigned to an object specifying which access modes (read, write, execute) are to be granted to processes with owner, group, or world access writes to the object.

**protection rings** A model of privilege separation consisting of a series of rings, with each successive ring representing greater execution privileges.

**pseudo device driver** In virtualization, a guest device driver that does not directly control system hardware but rather works with the virtual machine manager to access the device.

**PTE table** A Windows virtual-memory data structure.

**Pthreads** The POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization (a specification for thread behavior, not an implementation).

**public cloud** Cloud computing available via the Internet to anyone willing to pay for the services offered.

**public domain** The total absence of copyright protection. Software in the public domain can be used as desired by anyone, with no limits.

**public key** In asymmetric encryption algorithm, a key that can be distributed for encrypting and decrypting.

**public key encryption** A cipher algorithm in which different keys are used for encryption and decryption.

**pull migration** Migration that occurs when an idle processor pulls a waiting thread from a busy processor.

**pure demand paging** A demand paging scheme wherein no page is brought into memory until it is referenced.

**push** The action of placing a value on a stack data structure.

**push migration** Migration in which a task periodically checks the load on each processor and, if it finds an imbalance, evenly distributes the load by moving (or pushing) threads from overloaded to idle or less busy processors.

**Quest-V** An example of a partitioning hypervisor.

**queue** A sequentially ordered data structure that uses the first-in, first-out (FIFO) principle; items are removed from a queue in the order in which they were inserted.

**queueing-network analysis** An area of computing study in which algorithms are analyzed for various characteristics and effectiveness.

**race condition** A situation in which two threads are concurrently trying to change the value of a variable.

**RAID levels** The various types of RAID protection.

**RAM drives** Sections of a system's DRAM presented to the rest of the system as if they were secondary storage devices.

**random-access memory (RAM)** Rewritable memory, also called main memory. Most programs run from RAM, which is managed by the kernel.

**ransomware** A class of malware that disables computer access (frequently by encrypting files or the entire system) until a ransom is paid.

**rate** Generally, a measure of speed or frequency. A periodic real-time process has a scheduling rate of 1/p, where p is the length of its running period.

**rate-monotonic** A scheduling algorithm that schedules periodic tasks using a static priority policy with preemption.

**raw partition** A partition within a storage device not containing a file system.

**raw disk** Direct access to a secondary storage device as an array of blocks with no file system.

**raw I/O** Direct access to a secondary storage device as an array of blocks with no file system.

**read pointer** The location in a file from which the next read will occur.

**read-ahead** Sequential I/O performance optimization that reads and caches several subsequent pages when a read of one page is requested.

**readers-writers problem** A synchronization problem in which one or more processes or threads write data while others only read data.

**reader-writer lock** A lock appropriate for access to an item by two types of accessors, read-only and read-write.

**read-modify-write cycle** The situation in which a write of data smaller than a block requires the entire block to be read, modified, and written back.

**read-only memory (ROM)** A storage device whose contents are not modifiable.

**read-write (RW)** Access that allows reading and writing.

**ready queue** The set of processes ready and waiting to execute.

**real-time** A term describing an execution environment in which tasks are guaranteed to complete within an agreed-to time.

**real-time class** A scheduling class that segregates real-time threads from other threads to schedule them separately and provide them with their needed priority.

**real-time operating systems (RTOS)** Systems used when rigid time requirements have been placed on the operation of a processor or the flow of data; often used as control devices in dedicated applications.

**reapers** In memory management, routines that scan memory, freeing frames to maintain a minimum level of available free memory.

**recovery mode** A system boot state providing limited services and designed to enable the system admin to repair system problems and debug system startup.

**Red Hat** A popular Linux distribution.

**red-black tree** A tree containing n items and having at most lg n levels, thus ensuring worst-case performance of O(lg n).

**redirector** In Windows, a client-side object that forwards I/O requests to a remote system.

**redundant arrays of independent disks (RAID)** A disk organization technique in which two or more storage devices work together, usually with protection from device failure.

**reentrant code** Code that supports multiple concurrent threads (and can thus be shared).

**reference bit** An MMU bit indicating that a page has been referenced.

**reference string** A trace of accesses to a resource. In virtual memory, a list of pages accessed over a period of time.

**referenced pointer** In Windows, a means by which kernel-mode code can access objects; must be obtained by calling a special API.

**regions** In ARM v8 CPUs, contiguous areas of memory with separate privilege and access rules.

**registry** A file, set of files, or service used to store and retrieve configuration information. In Windows, the manager of hives of data.

**regressive round-robin** A variation on round-robin scheduling in which a thread that uses its entire CPU scheduling quantum is given a longer quantum and higher priority.

**relative access** A file-access method in which contents are read in random order, or at least not sequentially.

**relative block number** An index relative to the beginning of a file. The first relative block of the file is block 0, the next is block 1, and so on through the end of the file.

**relative path name** A path name starting at a relative location (such as the current directory).

**relocatable code** Code with bindings to memory addresses that are changed at loading time to reflect where the code is located in main memory.

**relocatable object file** The output of a compiler in which the contents can be loaded into any location in physical memory.

**relocation** An activity associated with linking and loading that assigns final addresses to program parts and adjusts code and data in the program to match those addresses.

**relocation register** A CPU register whose value is added to every logical address to create a physical address (for primitive memory management).

**remainder section** Whatever code remains to be processed after the critical and exit sections.

**remote access tool (RAT)** A back-door daemon left behind after a successful attack to allow continued access by the attacker.

**remote desktop** The representation of a desktop session to another system across a network, for remote access to the computer's GUI.

**remote desktop protocol (RDP)** A network protocol to allow remote access to a computer's display contents and keyboard and mouse input devices.

**remote file transfer** A function of a network operating system providing a means to transfer files between network-attached computers.

**remote procedure calls (RPCs)** Procedure calls sent across a network to execute on another computer; commonly used in client-server computing.

**remote-service mechanism** A facility, implemented by a feature such as RPC, in which clients ask a remote system to perform a function for them.

**renderer** A process that contains logic for rendering contents (such as web pages) onto a display.

**rendezvous** In interprocess communication, when blocking mode is used, the meeting point at which a send is picked up by a receive.

**replay attack** The malicious or fraudulent repetition of a valid transmission.

**replication** In file systems, the duplication and synchronization of a set of data over a network to another system. In storage, the automatic duplication of writes between separate sites.

**request edge** In a system resource-allocation graph, an edge (arrow) indicating a resource request.

**request manager** In Linux, the kernel component that manages the reading and writing of buffer contents to and from a block-device driver.

**resolve** Generally, to translate from a symbolic representation to a numeric one. In networking, to translate from a host name to a host-id. With files, to follow a link and find the target file.

**resource allocator** An operating system or application that determines how resources are to be used.

**resource manager** The role of an operating system in managing the computer's resources.

**resource sharing** The ability for multiple users, computers, etc., to access computing resources.

**resource utilization** The amount of a given resource (hardware or software) that is being used.

**response time** The amount of time it takes the system to respond to user action.

**restore** In file systems, the act of repairing or recovering files or a file system from a backup.

**resume** In virtualization, the continuation of execution after a guest's suspension.

**reverse engineering** The procedure of converting a compiled binary file into a human-readable format.

**rich text format** A file format developed by Microsoft that includes formatting details but can be used by various applications and operating systems, enabling files to be transferred between programs and systems.

**risk assessment** A systemic security analysis that attempts to value the assets of the entity in question and determine the odds that a security incident will affect the entity.

**roaming profile** In Windows, a collection of user preferences and settings that are kept on a server and allow a user's environment to follow that user from computer to computer.

**role-based access control (RBAC)** A method of access control in which roles rather than users have access rights; applies the principle of least privilege to the protection of operating systems.

**role** In RBAC, a named set of privileges that can be available to a user.

**root partition** The storage partition that contains the kernel and the root file system; the one mounted at boot.

**rotational latency** On an HDD, the time it takes the read-write head, once over the desired cylinder, to access the desired track.

**round-robin (RR)** A scheduling algorithm similar to FCFS scheduling, but with preemption added to enable the system to switch between threads; designed especially for time-sharing systems.

**router** A device or software that connects networks to each other (e.g., a home network to the Internet).

**RSA** The most widely used public key cipher.

**run queue** The queue holding the threads that are ready to run on a CPU.

**running** The state of the operating system after boot when all kernel initialization has completed and system services have started. In general, the system state after booting and before crashing or being shut down.

**run-time environment (RTE)** The full suite of software needed to execute applications written in a given programming language, including its compilers, libraries, and loaders.

**safe computing** Human behavior aimed at avoiding viruses and other security problems (e.g., by avoiding downloading pirated software).

**safe sequence** "In deadlock avoidance, a sequence of processes <P1, P2, ..., Pn> in which, for each Pi, the resource requests that Pi can make can be satisfied by the currently available resources plus the resources held by all Pj, with j < i."

**safe state** In deadlock avoidance, a state in which a system can allocate resources to each process in some order and still avoid deadlock.

**sandbox** A contained environment (e.g., a virtual machine).

**sandboxing** Restricting what an object can do by placing it in a contained environment (e.g., running a process on a virtual machine).

**SAS** A common type of I/O bus.

**scalability** Generally, the ability of a facility or service to increase capacity as needed by the users (e.g., to add more cores when the load increases).

**SCAN algorithm** An HDD I/O scheduling algorithm in which the disk head moves from one end of the disk to the other performing I/O as the head passes the desired cylinders; the head then reverses direction and repeats.

**scatter-gather** An I/O method in which multiple sources or destinations of I/O are specified in one command structure.

**scheduler** The part of the operating system that determines the next job to be done (e.g., the next process to be executed).

**scheduler activation** A threading method in which the kernel provides an application with a set of LWPs, and the application can schedule user threads onto an available virtual processor and receive upcalls from the kernel to be informed of certain events.

**scheduling classes** In Linux, classes on which scheduling is based; each class is assigned a specific priority.

**scheduling domain** A set of CPU cores that can be balanced against one another.

**scope** The time between when a lock is acquired and when it is released.

**script kiddie** An attacker who did not design the attack but instead is using an attack designed by a more sophisticated attacker.

**search path** In some operating systems, the sequence of directories searched for an executable file when a command is executed.

**second extended file system (ext2)** In Linux, an outdated version of the extended file system

**secondary storage** A storage system capable of holding large amounts of data permanently; most commonly, HDDs and NVM devices.

**second-chance page-replacement algorithm** A FIFO page replacement algorithm in which, if the reference bit is set, the bit is cleared and the page is not replaced.

**second-level interrupt handler** In some operating systems, the interrupt handler that actually handles interrupts; reception and queueing of interrupts are handled at another level (by the first-level handler).

**section object** The Windows data structure that is used to implement shared memory.

**sector forwarding** The replacement of an unusable HDD sector with another sector at some other location on the device.

**sector slipping** The renaming of sectors to avoid using a bad sector.

**sector sparing** The replacement of an unusable HDD sector with another sector at some other location on the device.

**sector** On an HDD platter, a fixed-size section of a track.

**secure** The state of a system whose resources are used and accessed as intended under all circumstances.

**secure by default** Describes a system or computer whose initial configuration decreases its attack surface.

**secure monitor call (SMC)** An ARM processor special instruction that can be used by the kernel to request services from the TrustZone.

**secure system process** In Windows, the process representing the fact that the secure kernel is loaded.

**security** The defense of a system from external and internal attacks. Such attacks include viruses and worms, denial-of-service attacks, identity theft, and theft of service.

**security access token** In Windows 10, a token created when a user logs in that contains the user's security ID, the security IDs of the groups the user belongs to, and a list of special privileges the user has.

**security context** In Windows 10, a characteristic, based on a user's access token, that enables a program run by the user to access what the user is allowed to access.

**security descriptor** In Windows 10, a feature that describes the security attributes of an object.

**security domain** The separation of systems and devices into classes, with each class having similar security needs.

**security ID (SID)** In Windows, a value used to uniquely identify a user or group for security purposes.

**security policy** A document describing the set of things being secured, how they are to be secured, and how users are to behave in matters relating to security.

**security reference monitor (SRM)** A Windows component that checks the effective security token whenever a thread opens a handle to a protected data structure.

**security through obscurity** A security layer in which information is kept private or obscured in the hope that it won't be discovered and used by attackers; an ineffective security method.

**security token** In Windows, a token associated with each process containing the SIDs of the user and the user's groups, the user's privileges, the integrity level of the process, the attributes and claims associated with the user, and any relevant capabilities.

**seek** The operation of changing the current file-position pointer.

**seek time** On an HDD, the time it takes the read-write head to position over the desired cylinder.

**semaphore** An integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal().

**semiconductor memory** The various types of memory constructed from semiconductors.

**sense key** In the SCSI protocol, information in the status register indicating an error.

**separation hypervisor** An experimental system that uses virtualization to partition separate system components into a chip-level distributed computing system.

**sequence number** In networking, a counter assigned to packets to order their assembly after delivery.

**sequential access** A file-access method in which contents are read in order, from beginning to end.

**serial-attached SCSI (SAS)** A common type of I/O bus.

**server** In general, any computer, no matter the size, that provides resources to other computers.

**server subject** In Windows 10 security, a process implemented as a protected server that uses the security context of the client when acting on the client's behalf.

**server system** A system providing services to other computers (e.g., a web server).

**server-message-block (SMB)** The Windows protocol for sending I/O requests over a network; a version was published as the common internet file system (CIFS).

**service** A software entity running on one or more machines and providing a particular type of function to calling clients. In Android, an application component with no user interface; it runs in the background while executing long-running operations or performing work for remote processes.

**service control manager (SCM)** In Windows 7, the component that manages services associated with plug-and-play devices.

**service-trigger** A mechanism in Windows 7 that allows plug-and-play device insertion to launch a service.

**session** In networking, a complete round of communication, frequently beginning with a login and ending with a logoff to terminate communications.

**session hijacking** The interception of a communication.

**session key** The TLS symmetric key, used for a web communication session, exchanged via asymmetric cryptography.

**SHA-1** An algorithm for creating a hash (a small, fixed-size block of data calculated from a larger data set, used to determine if a message has been changed).

**shared libraries** Libraries that can be loaded into memory once and used by many processes; used in systems that support dynamic linking.

**shared lock** A file lock similar to a reader lock in that several processes can obtain the lock concurrently.

**shared memory** In interprocess communication, a section of memory shared by multiple processes and used for message passing.

**shared system interconnect** A bus connecting CPUs to memory in such a way that all CPUs can access all system memory; the basis for NUMA systems.

**shared-memory model** An interprocess communication method in which multiple processes share memory and use that memory for message passing.

**shares** A basis for making scheduling decisions. The fair-share scheduling class uses CPU shares instead of priorities to allocate CPU time.

**shell** One of the command interpreters on a system with multiple command interpreters to choose from.

**shell script** A file containing a set series of commands (similar to a batch file) that are specific to the shell being used.

**shortest-job-first (SJF)** A scheduling algorithm that associates with each thread the length of the thread's next CPU burst and schedules the shortest first.

**shortest-remaining-time-first (SJRF)** A scheduling algorithm that gives priority to the thread with the shortest remaining time until completion.

**shortest-seek-time-first (SSTF) algorithm** An HDD I/O scheduling algorithm that sorts requests by the amount of seek time required to accomplish the request; the shortest time has the highest priority.

**shoulder surfing** Attempting to learn a password or other secret information by watching the target user at the keyboard.

**siblings** In a tree data structure, child nodes of the same parent.

**Siemens Jailhouse** An example of a partitioning hypervisor.

**signal** In UNIX and other operating systems, a means used to notify a process that an event has occurred.

**signature** In intrusion detection, a pattern of behavior associated with an attack.

**simple subject** In Windows 10 security, a subject that manages a user-initiated program's permissions.

**simultaneous multithreading (SMT)** The situation in which, in a CPU with multiple cores, each core supports multiple hardware threads.

**single indirect block** In UFS, a block containing pointers to direct blocks, which point to data blocks.

**single instruction multiple data (SIMD)** A form of parallelism in which multiple compute elements perform the same single instruction operating on multiple data points.

**single step** A CPU mode in which a trap is executed by the CPU after every instruction (to allow examination of the system state after every instruction); useful in debugging.

**single-threaded** A process or program that has only one thread of control (and so executes on only one core at a time).

**single-user mode** A system boot state providing limited services and designed to enable the system admin to repair system problems and debug system startup.

**Siri** The Apple voice-recognition system.

**sketch** An Arduino program.

**slab** A section of memory made up of one or more contiguous pages; used in slab allocation.

**slab allocation** A memory allocation method in which a slab of memory is allocated and split into chunks that hold objects of a given size. As the objects are freed, the chunks can coalesce into larger chunks, eliminating fragmentation.

**Slackware** An early but still widely used Linux distribution.

**slim reader-write lock (SRW)** A type of lock in modern Windows OS that favors neither readers nor writers.

**small computer-systems interface (SCSI)** One type of interface between a system and its storage (SCSI). See also ATA and SATA.

**snapshot** In file systems, a read-only view of a file system at a particular point in time; later changes do not affect the snapshot view.

**sniff** In network communication, to capture information by recording data as it is transmitted.

**sniffing** An attack in which the attacker monitors network traffic to obtain useful information.

**social engineering** A practice in which an attacker tricks someone into performing some task for the attacker (such as sending the attacker confidential information).

**socket** An endpoint for communication. An interface for network I/O.

**soft affinity** An operating system's policy of attempting to keep a process running on the same processor but not guaranteeing that it will do so.

**soft error** An error that is recoverable by retrying the operation.

**soft real-time systems** Systems that provide no guarantee as to when a critical real-time thread will be scheduled; they guarantee only that the thread will be given preference over noncritical threads

**Software as a Service (SaaS)** A type of computing in which one or more applications (such as word processors or spreadsheets) are available as a service via the Internet.

**software engineering** A field of study and a career involving writing software (i.e., programming.)

**software interrupt** A software-generated interrupt; also called a trap. The interrupt can be caused either by an error (e.g., division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed.

**software objects** The software components that make up a computer or device (files, programs, semaphores, etc.).

**software transactional memory (STM)** Transactional memory implemented exclusively in software; no special hardware is needed.

**Solaris** A UNIX derivative that is the main operating system of Sun Microsystems (now owned by Oracle Corporation). There is an active open source version called Illumos.

**Solaris ZFS** An advanced file system, first included as part of Solaris.

**solid-state disk** A disk-drive-like storage device that uses flash-memory-based nonvolatile memory.

**source file** A file containing the source code of a program.

**space sharing** A feature of APFS in which storage is treated as a pool and space is shared among the file systems created in that pool (much like ZFS).

**SPARC** A proprietary RISC CPU created by Sun Microsystems and now owned by Oracle Corporation. There is an active open source version called OpenSPARC.

**sparse** In memory management, a term describing a page table that has noncontiguous, scattered entries. A sparse address space has many holes.

**spinlock** A locking mechanism that continuously uses the CPU while waiting for access to the lock.

**split-screen** Running multiple foreground processes (e.g., on an iPad) but splitting the screen among the processes.

**spoof** The imitation of a legitimate identifier (such as an IP address) by an illegitimate user or system.

**spool** A buffer that holds output for a device (such as a printer) that cannot accept interleaved data streams.

**springboard** The iOS touch-screen interface.

**spyware** A Trojan horse variation in which the installed malware gathers information about a person or organization.

**stack** A sequentially ordered data structure that uses the last-in, first-out (LIFO) principle for adding and removing items; the last item placed onto a stack is the first item removed.

**stack algorithm** A class of page-replacement algorithms that do not suffer from Belady's anomaly.

**stack inspection** In Java, a protection procedure in which a calling sequence is checked to ensure that some caller in the sequence has been granted access to the resource called.

**stack section** The section of process memory that contains the stack; it contains activation records and other temporary data.

**stall** A CPU state occurring when the CPU is waiting for data from main memory and must delay execution.

**starvation** The situation in which a process or thread waits indefinitely within a semaphore. Also, a scheduling risk in which a thread that is ready to run is never put onto the CPU due to the scheduling algorithm; it is starved for CPU time.

**state** The condition of a process, including its current activity as well as its associated memory and disk contents.

**state information** In remote file systems, the set of information pertaining to connections and ongoing file operations (e.g., which files are open).

**state restore** Copying a process's context from its saved location to the CPU registers in preparation for continuing the process's execution.

**state save** Copying a process's context to save its state in order to pause its execution in preparation for putting another process on the CPU.

**stateless** In remote file systems, a protocol in which state need not be maintained for proper operation.

**static linking** Linking in which system libraries are treated like other object modules and combined by the loader into a binary program image.

**status register** A device I/O register in which status is indicated.

**storage-area network (SAN)** A local-area storage network allowing multiple computers to connect to one or more storage devices.

**stream cipher** A cipher that encrypts or decrypts a stream of bits or bytes (rather than a block).

**stream head** The interface between STREAMS and user processes.

**stream modules** In STREAMS, modules of functionality loadable into a STREAM.

**STREAMS** A UNIX I/O feature allowing the dynamic assembly of pipelines of driver code.

**stub** A small, temporary place-holder function replaced by the full function once its expected behavior is known.

**subject** In Windows 10 security, an entity used to track and manage user permissions.

**subsystem** A subset of an operating system responsible for a specific function (e.g., memory management).

**SunOS** The predecessor of Solaris by Sun Microsystems Inc.

**superblock** The UFS volume control block.

**superblock object** The VFS representation of the entire file system.

**supervisor mode** A CPU mode in which all instructions are enabled. The kernel runs in this mode. See also user mode.

**SuSE** A popular Linux distribution.

**suspend** In virtualization, to freeze a guest operating system and its applications to pause execution.

**swap map** In Linux swap-space management, a part of the data structure tracking swap-space use.

**swap space** Secondary storage backing-store space used to store pages that are paged out of memory.

**swapped** Moved between main memory and a backing store. A process may be swapped out to free main memory temporarily and then swapped back in to continue execution.

**swapping** Moving a process between main memory and a backing store. A process may be swapped out to free main memory temporarily and then swapped back in to continue execution.

**swap-space management** The low-level operating-system task of managing space on secondary storage for use in swapping and paging.

**symmetric clustering** A situation in which two or more hosts are running applications and are monitoring each other.

**symmetric encryption algorithm** A cryptography algorithm in which the same keys are used to encrypt and decrypt the message or data.

**symmetric multiprocessing (SMP)** Multiprocessing in which each processor performs all tasks, including operating-system tasks and user processes. Also, a multiprocessor scheduling method in which each processor is self-scheduling and may run kernel threads or user-level threads.

**synchronous** In interprocess communication, a mode of communication in which the sending process is blocked until the message is received by the receiving process or by a mailbox and the receiver blocks until a message is available. In I/O, a request that does not return until the I/O completes.

**synchronous threading** Threading in which a parent thread creating one or more child threads waits for them to terminate before it resumes.

**synchronous writes** Writes that are stored in the order in which they were issued, are not buffered, and have requesting threads wait for the writes to complete before continuing.

**system administrators** Computer users that configure, monitor, and manage systems.

**system build** Creation of an operating-system build and configuration for a specific computer site.

**system call** Software-triggered interrupt allowing a process to request a kernel service.

**system call** The primary interface between processes and the operating system, providing a means to invoke services made available by the operating system.

**system-call filtering** An operating-system facility to limit which system calls can be executed by a process.

**system daemon** A service that is provided outside the kernel by system programs that are loaded into memory at boot time and run continuously.

**system disk** A storage device that has a boot partition and can store an operating system and other information for booting the computer.

**system integrity protection (SIP)** A feature of macOS 10.11 and later versions that uses extended file attributes to mark system files as restricted so that even the root user cannot tamper with them.

**system mode** A CPU mode in which all instructions are enabled. The kernel runs in this mode. See also user mode.

**system process** A service that is provided outside the kernel by system programs that are loaded into memory at boot time and run continuously. In

Windows, a process that serves as the container of all internal kernel worker threads and other system threads created by drivers for polling, housekeeping, and other background work.

**system program** A program associated with the operating system but not necessarily part of the kernel.

**system resource-allocation graph** A directed graph for precise description of deadlocks.

**system restore point** In Windows, a copy of the system hives taken before any significant change is made to system configuration.

**system service** A collection of applications included with or added to an operating system to provide services beyond those provided by the kernel.

**system utility** A collection of applications included with or added to an operating system to provide services beyond what are provided by the kernel.

**system-call firewall** A firewall within a computer that limits the system calls a process can trigger.

**system-call interface** An interface that serves as the link to system calls made available by the operating system and that is called by processes to invoke system calls.

**system-contention scope (SCS)** A thread-scheduling method in which kernel-level threads are scheduled onto a CPU regardless of which process they are associated with (and thus contend with all other threads on the system for CPU time).

**system-development time** The time during which an operating system is developed, before it is made available in final "release" form.

**system-wide open-file table** A kernel in-memory data structure containing a copy of the FCB of each open file, as well as other information.

**target latency** In the Completely Fair Scheduler, a configurable variable which is the interval of time during which every runnable task should run at least once.

**targeted latency** An interval of time during which every runnable thread should run at least once.

**task control block** A per-process kernel data structure containing many pieces of information associated with the process.

**task parallelism** A computing method that distributes tasks (threads) across multiple computing cores, with each task is performing a unique operation.

**task** A process, a thread activity, or, generally, a unit of computation on a computer.

**templating** In virtualization, using one standard virtual-machine image as a source for multiple virtual machines.

**terabyte (TB)** 1,024^4 bytes.

**terminal concentrator** A type of front-end processor for terminals.

**tertiary storage** A type of storage that is slower and cheaper than main memory or secondary storage; frequently magnetic tape or optical disk.

**text file** A type of file containing text (alphanumeric characters).

**text section** The executable code of a program or process.

**thin client** A limited computer (terminal) used for web-based computing.

**third extended file system (ext3)** In Linux, a current version of the extended file system; the successor to ext2.

**thrashing** Paging memory at a high rate. A system thrashes when there is insufficient physical memory to meet virtual memory demand.

**thread** A process control structure that is an execution location. A process with a single thread executes only one task at a time, while a multithreaded process can execute a task per thread.

**thread cancellation** Termination of a thread before it has completed.

**thread dump** In Java, a snapshot of the state of all threads in an application; a useful debugging tool for deadlocks.

**thread library** A programming library that provides programmers with an API for creating and managing threads.

**thread pool** A number of threads created at process startup and placed in a pool, where they sit and wait for work.

**thread-environment block (TEB)** In Win32, a user-mode threads data structure that contains numerous per-thread fields.

**thread-local storage (TLS)** Data available only to a given thread.

**threat** The potential for a security violation.

**throughput** Generally, the amount of work done over time. In scheduling, the number of threads completed per unit time.

**tightly coupled systems** Systems with two or more processors in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices.

**time quantum** A small unit of time used by scheduling algorithms as a basis for determining when to preempt a thread from the CPU to allow another to run.

**time sharing** A practice in which the CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with the processes.

**time slice** A small unit of time used by scheduling algorithms as a basis for determining when to preempt a thread from the CPU to allow another to run.

**timer** A hardware component that can be set to interrupt the computer after a specified period.

**timestamp counter (TSC)** In Windows Vista, a counter that tracks execution time.

**TLB miss** A translation look-aside buffer lookup that fails to provide the address translation because it is not in the TLB.

**TLB reach** The amount of memory addressable by the translation look-aside buffer.

**TLB walk** The steps involved in walking through page-table structures to locate the needed translation and then copying that result into the TLB.

**touch screen** A touch-sensitive screen used as a computer input device.

**touch-screen interface** A user interface in which touching a screen allows the user to interact with the computer.

**trace tapes** A tool used in the evaluation of scheduling algorithms. Thread details are captured on real systems, and various algorithms are analyzed to determine their effectiveness.

**track** On an HDD platter, the medium that is under the read-write head during a rotation of the platter.

**transaction** Generally, the execution of a set of steps that make up one activity. In log-based transaction-oriented file systems, a set of operations completed as part of a request (e.g., “write this block to that file”).

**transactional memory** A type of memory supporting memory transactions.

**transfer rate** The rate at which data flows.

**translation granules** Features of ARM v8 CPUs that define page sizes and regions.

**translation look-aside buffer (TLB)** A small, fast-lookup hardware cache used in paged memory address translation to provide fast access to a subset of memory addresses.

**translation table base register** ARM v8 CPU register pointing to the level 0 (outer) page table for the current thread.

**transmission control protocol/Internet protocol (TCP/IP)** The most common network protocol; it provides the fundamental architecture of the Internet.

**transparent** In distributed computing, a term describing the automatic sharing of resources so that users do not know if a resource is local or remote.

**transport driver interface (TDI)** In Windows networking, an interface that supports connect-based and connectionless transports on top of the transport layer.

**transport layer security (TLS)** A cryptographic protocol that enables two computers to communicate securely; the standard protocol by which web browsers communicate to web servers.

**trap** A software interrupt. The interrupt can be caused either by an error (e.g., division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed.

**trap door** A back-door daemon left behind after a successful attack to allow continued access by the attacker.

**trap-and-emulate** In virtualization, a method used to implement virtualization on systems lacking hardware support (such as CPU instructions) for virtualization; any action that would cause the guest to call the operating system is intercepted, and the result is emulated.

**tree** A data structure that can be used to represent data hierarchically; data values in a tree structure are linked through parent–child relationships.

**triple DES** A modification of DES that uses the same algorithm three times and uses two or three keys to make the encryption more difficult to break.

**triple indirect block** In UFS, a block containing pointers to double indirect blocks, which point to single indirect blocks, which point to data blocks.

**Trojan horse** A program that acts in a clandestine or malicious manner rather than simply performing its stated function.

**TrustZone (TZ)** ARM processor implementation of the most secure protection ring.

**tunnel** In computer communication, a container of communications within another type of communication (e.g., a VPN that allows web traffic).

**turnstile** A Solaris scheduling feature using a queue structure containing threads blocked on a lock.

**two-factor authentication** Authentication based on two separate sources of data (e.g., a brain providing a password and a finger providing a fingerprint).

**type 0 hypervisor** A hardware-based virtualization solution that provides support for virtual machine creation and management via firmware (e.g., IBM LPARs and Oracle LDOMs).

**type 1 hypervisor** Operating-system-like software built to provide virtualization (e.g., VMware ESX, Joyent SmartOS and Citrix Xenserver).

**type 2 hypervisor** An application that runs on standard operating systems but provides virtual machine management features to guest operating systems (e.g., VMware workstation and fusion, and Oracle Virtualbox)

**type safety** In Java, a feature that ensures that classes cannot treat integers as pointers, write past the end of an array, or otherwise access memory in arbitrary ways.

**unbounded buffer** A buffer with no practical limit on its memory size.

**uncontended** A term describing a lock that is available when a thread attempts to acquire it.

**unified buffer cache** In file I/O, a cache used for both memory-mapped I/O and direct file I/O.

**unified extensible firmware interface (UEFI)** The modern replacement for BIOS containing a complete boot manager.

**unified virtual memory** In file I/O, the use of page caching for all types of I/O (explicit file system I/O and page fault I/O).

**uniform memory access (UMA)** Access to all main memory by all processors, without performance differences based on CPU or memory location.

**uniform naming convention (UNC)** A name format that includes the system and its resources (e.g.m \\server_name\share_name\x\y\z).

**unikernels** Specialized machine images that contain both an operating system and applications for efficient execution and increased security.

**universal serial bus (USB)** A type of I/O bus.

**universal Windows platform (UWP)** Windows 10 architecture that provides a common app platform for all devices that run it, including mobile devices.

**UNIX file system (UFS)** An early UNIX file systems; uses inodes for FCB.

**UnixBSD** A UNIX derivative based on work done at the University of California at Berkeley (UCB).

**unnamed semaphore** A POSIX scheduling construct that can only be used by threads in the same process.

**unstructured data** Data that are not in a fixed format (like a database record) but rather are free-form (like a twitter.com tweet).

**upcall** A threading method in which the kernel sends a signal to a process thread to communicate an event.

**upcall handler** A function in a process that handles upcalls.

**USB drive** Nonvolatile memory in the form of a device that plugs into a USB port.

**user** The human using a computer, or the identification of the human to the computer.

**user account** In Windows 10, an account belonging to a user (rather than a system account used by the computer).

**user authentication** The identification of a user of a computer.

**user datagram protocol (UDP)** A communications protocol layered on IP that is connectionless, is low latency, and does not guarantee delivery.

**user experience layer** In the layered macOS and iOS operating system design, the layer that defines the software interface that allows users to interact with computing devices.

**user file directory (UFD)** In two-level directory implementation, a per-user directory of files.

**user identifier (user ID) (UID)** A unique numerical user identifier.

**user interface (UI)** A method by which a user interacts with a computer.

**user mode** A CPU mode for executing user processes in which some instructions are limited or not allowed. See also kernel mode.

**user programs** User-level programs, as opposed to system programs.

**user rights** Permissions granted to users.

**user thread** A thread running in user mode.

**user-defined signal handler** The signal handler created by a process to provide non-default signal handling.

**user-initiated** In the Grand Central Dispatch Apple OS scheduler, the scheduling class representing tasks that interact with the user but need longer processing times than user-interactive tasks.

**user-interactive** In the Grand Central Dispatch Apple OS scheduler, the scheduling class representing tasks that interact with the user.

**user-mode driver framework (UMDF)** A framework in Windows to facilitate the writing of user-mode device drivers.

**user-mode scheduling (UMS)** A Microsoft Windows 7 feature that allows applications to create and manage threads independently of the kernel. This feature supports task-based parallelism by decomposing processes into tasks, which are then scheduled on available CPUs; it is used on AMD64 systems.

**user-mode thread (UT)** In Windows, the state of a thread when it is running in user mode.

**utility** In the Grand Central Dispatch Apple OS scheduler, the scheduling class representing tasks that require a longer time to complete but do not demand immediate results.

**utility storage** An inServ feature in which storage space can be increased as needed.

**valid-invalid** A page-table bit indicating whether a page-table entry points to a page within the logical address space of that process.

**variable-partition** A simple memory-allocation scheme in which each partition of memory contains exactly one process.

**vectored I/O** An I/O method in which multiple sources or destinations of I/O are specified in one command structure.

**version control system** Software that manages software distributions by allowing contributors to "push" changes into a repository and "pull" a version of the software source-code tree to a system (e.g., for compilation).

**victim frame** In virtual memory, the frame selected by the page-replacement algorithm to be replaced.

**view** In Windows, an address range mapped in shared memory. Also, the second step in memory-mapping a file, allowing a process to access the file contents.

**virtual address** An address generated by the CPU; must be translated to a physical address before it is used.

**virtual address control block (VACB)** The data structure in Windows that represents a cache block in the unified I/O cache.

**virtual address descriptor (VAD)** In Windows, a per-process descriptor of a virtual address range, kept in a tree data structure.

**virtual address space** The logical view of how a process is stored in memory.

**virtual CPU (VCPU)** In virtualization, a virtualized host CPU available to allocate to a guest operating system by the virtual machine manager.

**virtual file system (VFS)** The file-system implementation layer responsible for separating file-system-generic operations and their implementation and representing a file throughout a network

**virtual machine (VM)** The abstraction of hardware allowing a virtual computer to execute on a physical computer. Multiple virtual machines can run on a single physical machine (and each can have a different operating system).

**virtual machine control structures (VMCSs)** Hardware features provided by CPUs that support virtualization to track guest state.

**virtual machine manager (VMM)** The computer function that manages the virtual machine; also called a hypervisor.

**virtual machine sprawl** The situation in which there are so many virtual machines on a system that their use, history, and state become confusing and difficult to track; caused by the ease of creating virtual machines.

**virtual memory** A technique that allows the execution of a process that is not completely in memory. Also, separation of computer memory address space from physical into logical, allowing easier programming and larger name space.

**virtual memory fork** The vfork() system call, which forks a child process, suspends the parent, and lets the child share the parent's address space for both read and write operations (changes are visible to the parent).

**virtual private network (VPN)** An encrypted tunnel between two systems, commonly using IPSec, allowing secure remote access.

**virtual run time** A Linux scheduling aspect that records how long each task has run by maintaining the virtual run time of each task.

**virtual trust level (VTL)** A Windows 10 virtualization feature using Hyper-V to add more secure system modes.

**virtualization** A technology for abstracting the hardware of a single computer into several different execution environments, thereby creating the illusion that each environment is running on its own private computer.

**virtual-to-physical (V-to-P)** In virtualization, the conversion of a virtual machine guest to a physical system's operating system and applications.

**virus** A fragment of code embedded in a legitimate program that, when executed, can replicate itself; may modify or destroy files and cause system crashes and program malfunctions.

**virus dropper** The part of a virus that inserts the virus into the system.

**virus signature** A pattern that can be used to identify a virus within a system.

**VMware** Virtualization software company.

**VMware Workstation** A popular commercial type 2 hypervisor for x86 Windows systems.

**vnode** The virtual file system file representation structure, similar to the FCB for local files but applied to remote files.

**voice recognition** A computer interface based on spoken commands, which the computer parses and turns into actions.

**volatile** Describes storage whose content can be lost in a power outage or similar event.

**volatile storage** Storage whose content can be lost in a power outage or similar event.

**volume** A container of storage; frequently, a device containing a mountable file system (including a file containing an image of the contents of a device).

**volume control block** A per-volume storage block containing data describing the volume.

**von Neumann architecture** The structure of most computers, in which both process instructions and data are stored in the same main memory.

**VT-x** Intel x86 CPU virtualization-supporting instructions.

**wait queue** In process scheduling, a queue holding processes waiting for an event to occur before they need to be put on CPU.

**wait set** In Java, a set of threads, each waiting for a condition that will allow it to continue.

**wait-for graph** In deadlock detection, a variant of the resource-allocation graph with resource nodes removed; indicates a deadlock if the graph contains a cycle

**wear leveling** In nonvolatile memory, the effort to select all NAND cells over time as write targets to avoid premature media failure due to wearing out a subset of cells.

**wide-area network (WAN)** A network that links buildings, cities, or countries.

**WiFi** Wireless networking, consisting of devices and protocols that allow devices to attach to a network via radio waves rather than cables.

**Win32 API** The fundamental interface to the capabilities of Windows.

**Windows 10** A release of Microsoft Windows from 2009.

**Windows group policy** In Windows, a policy providing centralized management and configuration of operating systems, applications, and user settings in an Active Directory environment.

**Windows Subsystem for Linux (WSL)** A Windows 10 component allowing native Linux applications (ELF binaries) to run on Windows.

**Windows XP** A widely popular version of Microsoft Windows released in 2001.

**Winsock** The Windows socket API (similar to BSD sockets) for network communications.

**wired down** A term describing a TLB entry that is locked into the TLB and not replaceable by the usual replacement algorithm.

**wireless network** A communication network composed of radio signals rather than physical wires.

**witness** A lock order verifier.

**word** A unit made up of one or more bytes. For example, a computer that has 64-bit registers and 64-bit memory addressing typically has 64-bit (8-byte) words.

**working set** The set of pages in the most recent page references.

**working-set maximum** The maximum number of frames allowed to a process in Windows.

**working-set minimum** The minimum number of frames guaranteed to a process in Windows.

**working-set model** A model of memory access based on tracking the set of most recently accessed pages.

**working-set window** A limited set of most recently accessed pages (a "window" view of the entire set of accessed pages).

**workstation** A powerful personal computer (PC) for engineering and other demanding workloads.

**world rights** A category of file access rights.

**World Wide Web (WWW)** The Internet; a worldwide system of interconnected computer networks.

**WORM** Write-once, read-many-times storage.

**worm** A program that spreads malware between computers without intervention from humans.

**worst-fit** In memory allocation, selecting the largest hole available.

**write amplification** The creation of I/O requests not by applications but by NVM devices doing garbage collection and space management, potentially impacting the devices' write performance.

**write pointer** The location in a file to which the next write will occur.

**write-anywhere file layout (WAFL)** The file system that is the heart of the NetApp, Inc., storage appliances.

**write-back caching** In caching, a policy whereby data are first written to the cache; later, the cache writes the change to the master copy of the data.

**write-on-close policy** In caching, a policy whereby writes reside in the cache until the file is closed and are only then written back to the master copy of the data.

**write-through policy** In caching, a policy whereby writes are not cached but are written through the cache to the master copy of the data.

**x86-64** A class of 64-bit CPUs running an identical instruction set; the most common CPUs in desktop and server systems.

**Xen** Virtualization software company.

**XML firewall** A firewall that examines and limits XML traffic.

**Xtratum** An example of a partitioning hypervisor.

**yellow pages** A distributed naming service that provides username, password, hostname, and printer information to a set of computers.

**zero-day attacks** Attacks that have not been seen before and therefore cannot be detected via their signatures.

**zero-fill-on-demand** The writing of zeros into a page before it is made available to a process (to keep any old data from being available to the process).

**ZFS** Oracle file system, created by Sun Microsystems, with modern algorithms and features and few limits on file and device sizes.

**zombie** A process that has terminated but whose parent has not yet called wait() to collect its state and accounting information.

**zombie systems** Compromised systems that are being used by attackers without the owners' knowledge.

**zones** In application containment, a virtual layer between the operating system and a process in which the application runs, limiting its normal access to system resources. In Linux, the four regions of kernel memory.

# WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley’s ebook EULA.